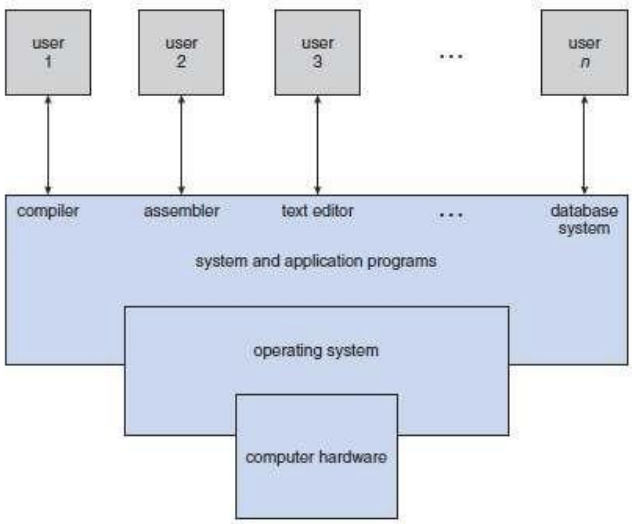
MODULE 1: INTRODUCTION

An **operating system** is a program that manages the computer hardware.
It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware.
Some OS designed for

- *efficient* -Mainframe operating systems are designed primarily to optimize utilization of hardware.
- *convenient* -Personal computer (PC) operating systems support complex games, business applications, and everything in between.
- *Convenient & efficient*- Operating systems for handheld computers are designed to provide an environment in which a user can easily interface with the computer to execute programs.

1.1. What Operating Systems do? (Role of OS)

A computer system can be divided roughly into Four components of a computer (as shown in figure):
1) Hardware
2) OS
3) Application programs and
4) Users



• Hardware provides basic computing-resources:
→ CPU
→ memory and
→ I/O devices.

• Application-program defines how the resources are used to solve computing-problems of the users. Ex: word processors, spread sheets, compilers.

• The OS controls & co-ordinates the use of hardware among various application-program for various users.

• Two views of OS: 1) User and
2) System.

1.1.1 User View

• The user's view of the computer depends on the interface being used:

Case 1: Most users use a PC consisting of a monitor, keyboard and system-unit.
The OS is designed mostly for ease of use.
-Some attention is paid to performance.
-No attention is paid to resource utilization.
-The OS is optimized for the single-user experience.

Case 2: Some users use a terminal connected to a mainframe or (a minicomputer).
The OS is designed
→ to maximize resource utilization.
→ to assure that no individual user takes more than her fair share.

Case 3: Some users use a workstation connected to network.

-The users have dedicated resources such as networking and servers.

-The OS is designed to compromise between

→ individual usability and

→ resource utilization.

Case 4: Some users use a handheld computer.

-These might be connected to networks, either directly by wire or (more often) through wireless modems and networking.

-Because of power, speed, and interface limitations, they perform relatively few remote operations.

- The OS is designed mostly for individual usability.

- Performance per unit of battery life is a very important factor.

Case 5: Some computers have little or no user view.

-For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

1.1.2 System View

1) An OS as a resource allocator

Resources used to solve a computing-problem:

→ CPU time

→ memory-space

→ file-storage space and

→ I/0 devices.

The OS manages and allocates the above resources to programs and the users.

2) An OS as a control program

-The OS is needed to control:

→ operations of I/0 devices and

→ execution of user-programs to prevent errors.

→ improper use of computer resourses

1.1.3 Defining Operating Systems

-no universally accepted definition

Definition 1:

Operating systems exist because they offer a reasonable way to solve the problem of creating a usable computing system.

The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. Toward this goal, computer hardware is constructed.

Since bare hardware alone is not particularly easy to use, application programs are developed.

These programs require certain common operations, such as those controlling the I/O devices.

The common functions of controlling and allocating resources are then brought together into one piece of software: the operating system.

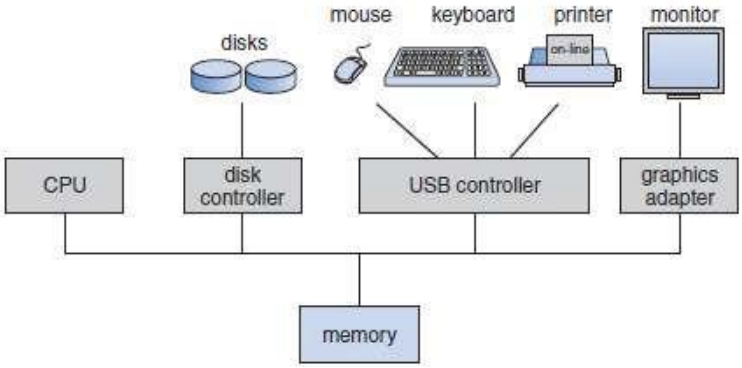
Definition 2(generally followed):

A more common definition is that the operating system is the one program running at all times on the computer (usually called the **kernel),** with all else being systems programs and application programs.

1.2 Computer-System Organization

1.2.1 Computer-System Operation

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common bus that provides access to shared memory (as shown in Figure).



- Each device controller is in charge of a specific type of device (for example, disk drives, audio devices, and video displays).
- The CPU and the device controllers can execute concurrently, competing for memory cycles.
- To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory.

- For a computer to start running

when it is powered up or rebooted—it needs to have an initial program to run.
This initial program, or **bootstrap program**
-it is stored in read-only memory (ROM)
-It initializes all aspects of the system, from CPU registers to device controllers to memory contents.
-The bootstrap program must know how to load the operating system and to start executing that system.
-To accomplish this goal, the bootstrap program must locate and load into memory the operating system kernel.
-The operating system then starts executing the first process, such as "init," and waits for some event to occur.

- The occurrence of an event is usually signaled by an **interrupt** from -either the hardware or the software.

-Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
-Software may trigger an interrupt by executing a special operation called a **system call** (also called a **monitor call).**

- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location.

-The fixed location usually contains the starting address where the service routine for the interrupt is located.
-The interrupt service routine executes; on completion, the CPU resumes the interrupted computation.

- Interrupts are an important part of a computer architecture.
- The interrupt must transfer control to the appropriate interrupt service routine.
- methods for handling interrupt:
    1) to invoke a generic routine to examine the interrupt information; the routine, in turn,would call the interrupt-specific handler.
    2) Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines can be used to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed.

-Generally, the table of pointers is stored in low memory.
-These locations hold the addresses of the interrupt service routines for the various devices.
-This array, or **interrupt vector,** of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.

- The interrupt architecture must also save the address of the interrupted instruction.
  - Many old designs simply stored the interrupt address in a fixed location or in a location indexed by the device number.
  - More recent architectures store the return address on the system stack.
    - If the interrupt routine needs to modify the processor state—for instance, by modifying register values—it must explicitly save the current state and then restore that state before returning.
    - After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred

1.2.2 Storage Structure

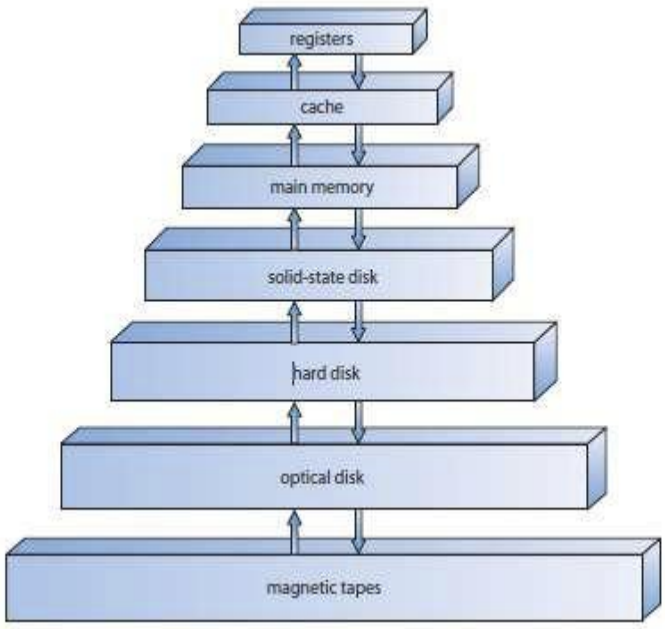
• Programs must be in main-memory (also called RAM) to be executed.
• Interaction with memory is done through a series of load or store instructions.

1) Load Instruction
 Moves a word from main-memory to an internal register within the CPU.
2) Store Instruction
 Moves the content of a register to main-memory.

• Also, the CPU automatically loads instructions from main-memory for execution.
• Ideally, we want the programs & data to reside in main-memory permanently. This is not possible for 2 reasons:

1) Main-memory is small.
2) Main-memory is volatile i.e. it loses its contents when powered-off.

• Most computers provide secondary-storage as an extension of main-memory.
For ex: magnetic disk.
• Main requirement:
-The secondary-storage must hold large amount of data permanently.
• The wide range of storage-systems can be organized in a hierarchy (as shown in Figure).
• The higher levels are expensive, but they are fast.
• The lower levels are inexpensive, but they are slow.



1.2.3 I/O Structure

• A computer consists of CPUs and multiple device controllers (as shown in figure).

• A device controller is in charge of a specific type of device.

• The device controller maintains

→ some local buffer and

→ set of special-purpose registers.

• Typically, OS has a device-driver for each controller.

• The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage

• to start an I/O operation :

1) Driver loads the appropriate registers within the controller.

2) Controller examines the contents of registers to determine what action to take.

3) Controller transfers data from the device to its local buffer.

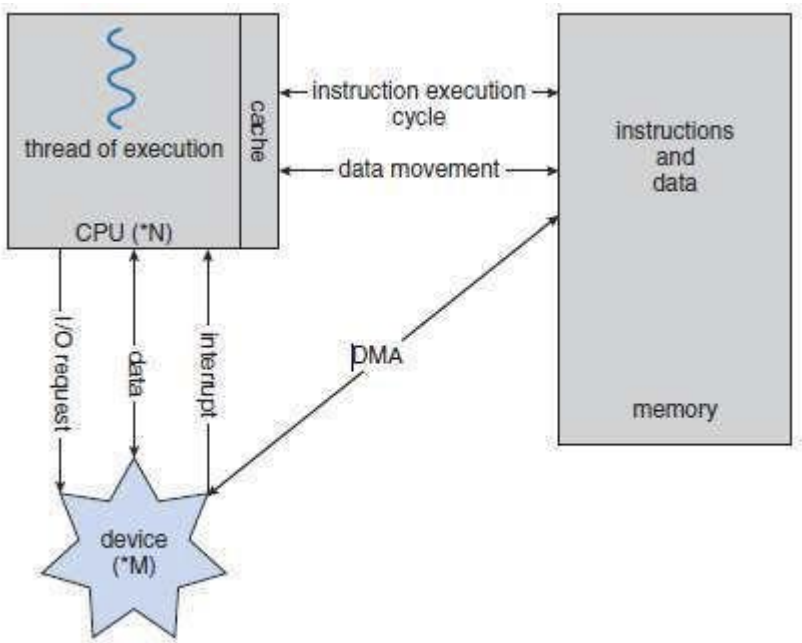
4) Controller informs the driver via an interrupt that it has finished its operation.

5) Driver then returns control to the OS.

• Problem: Interrupt-driven I/O produces high overhead when used for bulk data-transfer.

Solution: Use DMA (direct memory access).

• In DMA, the controller transfers blocks of data from buffer-storage directly to main memory without CPU intervention (illustrated in figure)



1.3. Computer System Architecture

A computer system may be organized in a number of different ways, which we can categorize roughly according to the number of general-purpose processors used.

1) Single-Processor Systems
2) Multiprocessor Systems
3) Clustered Systems

1.3.1 Single Processor Systems

• The system has only one general-purpose CPU.
• The CPU is capable of executing a general-purpose instruction-set.
• These systems range from PDAs through mainframes.
• Almost all systems have following processors:

1) Special Purpose Processors

-Include disk, keyboard, and graphics controllers.
-Special-purpose processors run a limited instruction set and do not run user processes.
-Sometimes they are managed by the operating system and sometimes not.

2) General Purpose Processors

- Include I/O processors.

• The use of special-purpose microprocessors is common and does not turn a single-processor system into a multiprocessor.
• If there is only one general-purpose CPU, then the system is a single-processor system.

1.3.2 Multi-Processor Systems

• These systems have two or more processors which can share:
→ bus
→ clock
→ memory/peripheral devices
• Advantages:

1) Increased Throughput

- By increasing no. of processors, we expect to get more work done in less time.
-The speed-up ratio with *N* processors is not *N*, however; rather, it is less than N. When multiple processors -cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly.
- This overhead, plus contention for shared resources, lowers the expected gain from additional processors.

2) Economy of Scale

- These systems are cheaper because they can share
→ peripherals
→ mass-storage
→ power-supply.
- If many programs operate on same data, they will be stored on one disk & all processors can share them.

3) Increased Reliability

- The failure of one processor will not halt the system.
-Increased reliability of a computer system is crucial in many applications.
-The ability to continue providing service proportional to the level of surviving hardware is called **graceful degradation.**
-Some systems go beyond graceful degradation and are called **fault tolerant,** because they can suffer a failure of any single component and still continue operation.
-The fault tolerance requires a mechanism to allow the failure to be detected, diagnosed, and, if possible, corrected. Example : HP NonStop

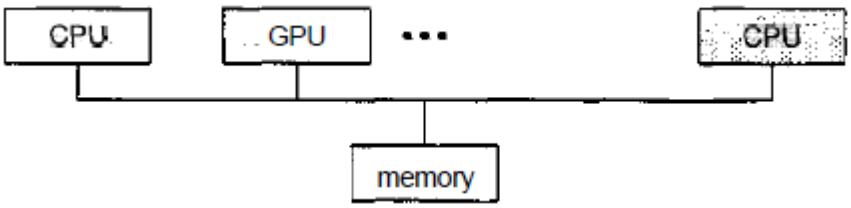
Two types of multiple-processor systems: 1) Asymmetric multiprocessing (AMP) and
2) Symmetric multiprocessing (SMP)

1) Asymmetric Multiprocessing
- This uses master-slave relationship
- Each processor is assigned a specific task.
- A master-processor controls the system.

The other processors look to the master for instruction.
- The master-processor schedules and allocates work to the slave-processors.

2) Symmetric Multiprocessing
- Each processor runs an identical copy of OS.
- All processors are peers; no master-slave relationship exists between processors as shown in figure.
- **Advantages:** Many processes can run simultaneously.
- **Disadvantage:** Since CPUs are separate, one CPU may be sitting idle while another CPU is overloaded. This results in inefficiencies.

Solution: Processes and resources are shared dynamically among the various processors



- A recent trend in CPU design is to include multiple compute **cores** on a single chip. In essence, these are multiprocessor chips.

-N-way chips are common in high-end systems.
-these multi-core CPUs look to the operating system just as N standard processors.

- **blade** server: multiple processor boards, I/O boards, and networking boards are placed in the same chassis.

The difference between these and traditional multiprocessor systems is that each blade-processor board boots independently and runs its own operating system.

1.3.3 Clustered Systems

- These systems consist of two or more systems coupled together (Figure 1.7).
- These systems share storage & closely linked via LAN.
- Advantage:
  1) Used to provide high-availability service.
- High-availability is obtained by adding a level of redundancy in the system.
- To provide shared access to data, the system must also supply access control and locking to ensure that no conflicting operations occur. This function, commonly known as a **distributed lock manager (DLM)**

• Working procedure:
- A cluster-software runs on the cluster-nodes.
- Each node can monitor one or more other nodes (over the LAN).
- If the monitored-node fails, the monitoring-node can
→ take ownership of failed-nodes storage and
→ restart the applications running on the failed-node.
-The users and clients of the applications see only a brief interruption of service.

• Two types are: 1) Asymmetric Clustering and
2) Symmetric Clustering

1) Asymmetric Clustering
• One node is in hot-standby mode while the other nodes are running the applications.
• The hot-standby node does nothing but monitor the active-server.
• If the server fails, the hot-standby node becomes the active server.

2) Symmetric Clustering
• Two or more nodes are running applications, and are monitoring each other.
• Advantage:
1) This mode is more efficient, as it uses all of the available hardware.
• It does require that more than one application be available to run.

1.4 Operating System Structure

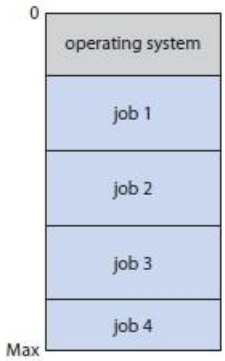
1) Batch Systems
2) Multi-Programmed Systems
3. Time-Sharing Systems

1.4.1 Batch Systems

• Early computers were physically enormous machines run from a console.
• The common input devices were card readers and tape drives.
• The common output devices were line printers, tape drives, and card punches.
• The user
→ prepared a job which consisted of the program, the data, and control information
→ submitted the job to the computer-operator.
• The job was usually in the form of punch cards.
• At some later time (after minutes, hours, or days), the output appeared.
• To speed up processing, operators batched together jobs with similar needs and ran them through the computer as a group.
• Disadvantage:
1) The CPU is often idle, because the speeds of the mechanical I/O devices.

1.4.2 Multi-Programmed Systems

• Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute.



• The idea is as follows:

1) OS keeps several jobs in memory simultaneously as shown in figure.
2) OS picks and begins to execute one of the jobs in the memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete.
3) OS simply switches to, and executes, another job.
4) When that job needs to wait, the CPU is switched to another job, and so on.
5) As long as at least one job needs to execute, the CPU is never idle.

• If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision is job scheduling.

• If several jobs are ready to run at the same time, the system must choose among them. Making this decision is CPU scheduling.

1.4.3 Time Sharing Systems

• Time sharing (or multitasking) is a logical extension of multiprogramming.
• The CPU executes multiple jobs by switching between them.
• Switching between jobs occur so frequently that the users can interact with each program while it is running.
• Many users are allowed to share the computer simultaneously.
• CPU scheduling and multiprogramming are used to provide each user with a small portion of a timeshared computer.
• To obtain a good response time, jobs may have to be swapped in and out of main memory to the disk (called as backing store).
• Solution: Virtual memory
• Virtual memory is a technique that allows the execution of a process that may not be completely in memory.
• Advantage of virtual-memory:

1) Programs can be larger than physical memory.

• Main requirements:

-The system must provide a file-system.
- The system must provide disk-management.
- The system must provide CPU-scheduling to support concurrent execution.
- The system must provide job-synchronization to ensure orderly execution.

1.5. Operating System Operations

• Modern OS is interrupt driven.
• Events are always signaled by the occurrence of an interrupt or a trap.
• A trap is a software generated interrupt caused either by
→ error (for example division by zero) or
→ request from a user-program that an OS service be performed.
• For each type of interrupt, separate segments of code in the OS determine what action should be taken.
• ISR (Interrupt Service Routine) is provided that is responsible for dealing with the interrupt.

1.5.1 Dual Mode Operation : to ensure proper execution of OS

• Problem: We must be able to differentiate between the execution of

→ OS code and

→ user-defined code.

Solution: Most computers provide hardware-support.

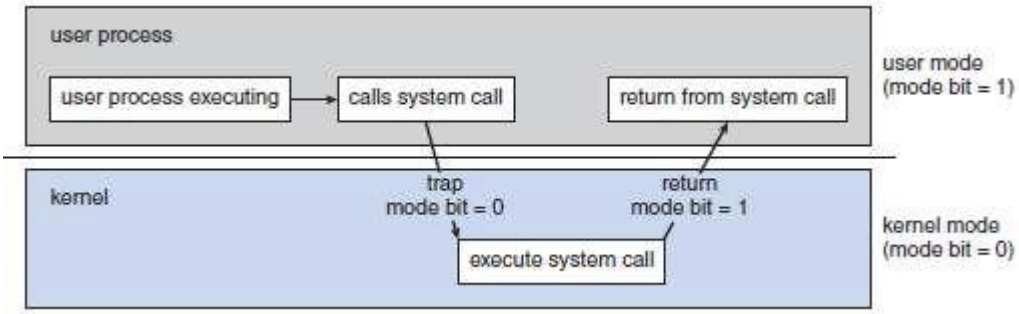
• Two modes of operation :1) User mode and

2) Kernel mode (also called supervisor mode, system mode, or privileged mode)

• A mode bit is a bit added to the hardware of the computer to indicate the current mode:

i.e. kernel (0) or user (1)

Figure below shows Transition from user to kernel mode



• Working principle:

1) At system boot time, the hardware starts in kernel-mode.

2) The OS is then loaded and starts user applications in user-mode.

3) Whenever a trap or interrupt occurs, the hardware switches from user-mode to kernel-mode (that is, changes the state of the mode bit to 0).

4) The system always switches to user-mode (by setting the mode bit to 1) before passing control to a user-program.

• Dual mode protects

→ OS from errant users and

→ Errant users from one another.

• Privileged instruction is executed only in kernel-mode.

• If an attempt is made to execute a privileged instruction in user-mode, the hardware treats it as illegal and traps it to the OS.

• A system calls are called by user-program to ask the OS to perform the tasks on behalf of the user program.

1.5.2 Timer

• **Problem**: We cannot allow a user-program to get stuck in an infinite loop and never return control to the OS.

Solution: We can use a timer.

• A timer can be set to interrupt the computer after a specific period.

• The period may be fixed (for ex: 1/60 second) or variable (for ex: from 1ns to 1ms).

• A variable timer is implemented by a fixed-rate clock and a counter.

• Working procedure:

1) The OS sets the counter.

2) Every time the clock ticks, the counter is decremented.

3) When the counter reaches 0, an interrupt occurs, then the OS either terminates or gives extra time.

• The instructions that modify the content of the timer are privileged instructions.

1.6. Process Management

• A program does nothing unless its instructions are executed by a CPU.
• A program in execution is a process.
• A process needs following resources to do a task:
→ CPU
→ memory and
→ files.
• The resources are allocated to process
→ when the process is created or
→ while the process is running.
• When the process terminates, the OS reclaims all the reusable resources.
• A program by itself is not a process;

1) A program is a passive entity (such as the contents of a file stored on disk).
2) A process is an active entity.

• Two types of process:

1) Single-threaded process has one PC(program counter) which specifies location of the next instruction to be executed.
2) Multi-threaded process has one PC per thread which specifies location of next instruction to execute in each thread

• The OS is responsible for the following activities:

1) Creating and deleting both user and system processes
2) Suspending and resuming processes
3) Providing mechanisms for process synchronization
4) Providing mechanisms for process communication
5) Providing mechanisms for deadlock handling

1.7 Memory Management

• Main memory is the array of bytes ranging from hundreds to billions.
• Each byte has its own address.
• The CPU

→ reads instructions from main memory during the instruction-fetch cycle.
→ reads/writes data from/to main-memory during the data-fetch cycle.

• To execute a program:

1) The program will be
→ loaded into memory and
→ mapped to absolute addresses.
2) Then, program accesses instructions & data from memory by generating absolute addresses.
3) Finally, when program terminates, its memory-space is freed.

• To improve CPU utilization, keep several programs will be kept in memory
• Selection of a memory-management scheme depends on hardware-design of the system.
• The OS is responsible for the following activities:

1) Keeping track of which parts of memory are currently being used and by whom
2) Deciding which processes are to be loaded into memory when memory space becomes available
3) Allocating and de-allocating memory space as needed.

1.8 Storage Management

To make the computer system convenient for users, the operating system provides a uniform, logical view of information storage

1) File-System Management
2) Mass-Storage Management
3) Caching

1.8.1 File System Management

• Computer stores information on different types of physical media.
For ex: magnetic disk, optical disk.
• Each medium is controlled by a device (e.g. disk drive).
• The OS
→ maps files onto physical media and
→ accesses the files via the storage devices
• File is a logical collection of related information.
• File consists of both program & data.
• Data files may be numeric, alphabets or binary.
• When multiple users have access to files, access control (read, write) must be specified.
• The OS is responsible for following activities:
1) Creating and deleting files.
2) Creating and deleting directories.
3) Supporting primitives for manipulating files & directories.
4) Mapping files onto secondary storage.
5) Backing up files on stable (non-volatile) storage media.

1.8.2 Mass Storage Management

• Usually, disks used to store
→ data that does not fit in main memory or
→ data that must be kept for a "long" period of time.
• Most programs are stored on disk until loaded into memory.
• The programs include
→ compilers
→ word processors and
→ editors.
• These programs use the disk as both the source and destination of their processing.
• Entire speed of computer operation depends on disk and its algorithms.
•There are, however, many uses for storage that is slower and lower in cost (and sometimes of higher capacity) than secondary storage.
•Backups of disk data, seldom-used data, and long-term archival storage are some examples.
•Magnetic tape drives and their tapes and CD and DVD drives and platters are typical **tertiary storage** devices.
•Tertiary storage is not crucial to system performance, but it still must be managed
• The OS is responsible for following activities:
1) Free-space management
2) Storage allocation and
3) Disk scheduling

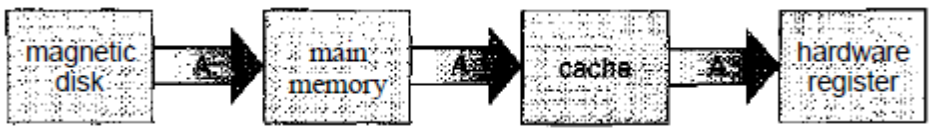
1.8.3 Caching

• Caching is an important principle of computer systems.
• Information is normally kept in some storage system (such as main memory).
• As it is used, it is copied into a faster storage system called as the cache on a temporary basis.
• When we need a particular piece of information:

1) We first check whether the information is in the cache.
2) If information is in cache, we use the information directly from the cache.
3) If information is not in cache, we use the information from the source, putting a copy in the cache under the assumption that we will need it again soon.

• In addition, internal programmable registers, such as index registers, provide high-speed cache for main memory.
• The compiler implements the register-allocation and register-replacement algorithms to decide which information to keep in registers and which to keep in main memory.
• Most systems have an instruction cache to hold the instructions expected to be executed next.
• Most systems have one or more high-speed data caches in the memory hierarchy
• Because caches have limited size, cache management is an important design problem
• Careful selection of cache size & of a replacement policy can result in greatly increased performance
• The movement of information between levels of a storage hierarchy may be either explicit or implicit, depending on the hardware design and the controlling operating-system software.
• In a hierarchical storage structure, the same data may appear in different levels of the storage system.
• For example,

• suppose that an integer A that is to be incremented by 1 is located in file B, and file B resides on magnetic disk.
• The increment operation proceeds by first issuing an I/O operation to copy the disk block on which A resides to main memory.
• This operation is followed by copying A to the cache and to an internal register.
• Thus, the copy of A appears in several places: on the magnetic disk, in main memory, in the cache, and in an internal register (as shown in Figure).



• Once the increment takes place in the internal register, the value of A differs in the various storage systems.
• The value of A becomes the same only after the new value of A is written from the internal register back to the magnetic disk.

•In a **computing environment where only one process** executes at a time,this arrangement poses no difficulties, since an access to integer A will always be to the copy at the highest level of the hierarchy.

• In **a multitasking environment**, where the CPU is switched back and forth among various processes, extreme care must be taken to ensure that, if several processes wish to access A, then each of these processes will obtain the most recently updated value of A.

• It is more complicated in a **multiprocessor environment** where, in addition to maintaining internal registers, each of the CPUs also contains a local cache. In such an environment, a copy of A may exist simultaneously in several caches. Since the various CPUs can all execute concurrently, we must make sure that an update to the value of A in one cache is immediately reflected in all other caches where A resides. This situation is called **cache coherency.**

• In a **distributed environment**, the situation becomes even more complex. In this environment, several copies (or replicas) of the same file can be kept on different computers that are distributed in space. Since the various replicas may be accessed and updated concurrently, some distributed systems ensure that, when a replica is updated in one place, all other replicas are brought up to date as soon as possible.

1.8.4 I/O Systems

• The OS must hide peculiarities of hardware devices from users.
• In UNIX, the peculiarities of I/O devices are hidden from the bulk of the OS itself by the I/O subsystem.
• The I/O subsystem consists of
1) A memory-management component that includes buffering, caching, and spooling.
2) A general device-driver interface.
3) Drivers for specific hardware devices.
• Only the device driver knows the peculiarities of the specific device to which it is assigned.

1.9 Protection and Security

• Protection is a mechanism for controlling access of processes or users to resources defined by OS.
• This mechanism must provide
→ means for specification of the controls to be imposed and → means for enforcement.
• Protection can improve reliability by detecting latent errors at the interfaces between subsystems.
• Security means defense of the system against internal and external attacks.
• The attacks include
→ viruses and worms
→ DOS(denial-of-service)
→ identity theft.
• Protection and security require the system to be able to distinguish among all its users.
• Most operating systems maintain a list of user names and associated **user identifiers (user IDs).**
• That user ID is associated with all of the user's processes and threads.
• In some circumstances, we wish to distinguish among sets of users rather than individual users.
• To accomplish this, we need to define a group name and the set of users belonging to that group.
• Group functionality can be implemented as a system-wide list of group names and **group identifiers.**
• A user can be in one or more groups, depending on operating-system design decisions.
• The user's group IDs are also included in every associated process and thread.
• a user sometimes needs to **escalate privileges** to gain extra permissions for an activity.
• Operating systems provide various methods to allow privilege escalation.

1.10 Distributed System

• This is a collection of physically separate, possibly heterogeneous computer-systems.
• The computer-systems are networked to provide the users with access to the various resources.
• Access to a shared resource increases
→ computation speed
→ functionality
→ data availability and → reliability
• A network is a communication path between two or more systems.

• Networks vary by the
→ protocols used
→ distances between nodes and
→ transport media.
• Common network protocol are
→ TCP/IP
→ ATM.
• Networks are characterized based on the distances between their nodes.
→ A local-area network (LAN) connects computers within a building.
→ A wide-area network (WAN) usually links buildings, cities, or countries.
→ A metropolitan-area network (MAN) could link buildings within a city.
→small-area network-few feets using Bluetooth
• The media to carry networks are equally varied. They include
→ copper wires,
→ fiber strands, and
→ wireless transmissions.

• A **network operating system** is an operating system that provides features such as file sharing across the network and that includes a communication scheme that allows different processes on different computers to exchange messages.

• A computer running **a network operating system** acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers.

• **A distributed operating system** provides a less autonomous environment: The different operating systems communicate closely enough to provide the illusion that only a single operating system controls the network.

1.11. Special Purpose Systems

There are different classes of computer systems whose

-functions are more limited and

-whose objective is to deal with limited computation domains.

1) Real-Time Embedded Systems
2) Multimedia Systems
3) Handheld Systems

1.11.1 Real-Time Embedded Systems

- Embedded computers are the most prevalent form of computers in existence.
- These devices are found everywhere, from car engines and manufacturing robots to VCRs and microwave ovens.
- They tend to have very specific tasks.
- The systems they run on are usually primitive, and so the operating systems provide limited features.
- Usually, they prefer to spend their time monitoring & managing hardware devices such as
  → automobile engines and
  → robotic arms.
- Embedded systems almost always run real-time operating systems.
- A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application.
- Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs.
- A real-time system has well-defined, fixed time constraints.
- Processing *mustbe* done within the defined constraints, or the system will fail.
- A real-time system functions correctly only if it returns the correct result within its time constraints.
- In contrast time-sharing system may have no time constraints at all.

1.11.2 Multimedia Systems

• Multimedia data consist of audio and video files as well as conventional files.

• These data differ from conventional data in that multimedia data must be delivered(streamed) according to certain time restrictions.

• Multimedia describes a wide range of applications. These include

→ audio files such as MP3

→ DVD movies

→ video conferencing

→ live webcasts of speeches

• Multimedia applications need not be either audio or video; rather, a multimedia application often includes a combination of both

1.11.3 Handheld Systems

•Handheld systems include

→ PDAs and

→ cellular telephones

• Main challenge faced by developers of handheld systems: Limited size of devices.

• Because of small size, most handheld devices have a
→ small amount of memory,
-As a result, the operating system and applications must manage memory efficiently.
-This includes returning all allocated memory back to the memory manager when the memory is not being used
-Currently, not many handheld devices use virtual memory techniques, so program developers must work within the confines of limited physical memory.

→slow processors,
-Faster processors requiremore power.
-To include a faster processor in a handheld device would require a larger battery, which would take up more space and would have to be replaced (or recharged) more frequently.
-Most handheld devices use smaller, slower processors that consume less power.
-Therefore, the operating system and applications must be designed not to tax the processor.

→small display screens.
-A lack of physical space limits input methods to small keyboards, handwriting recognition, or small screen-based keyboards.
-The small display screens limit output options
-Familiar tasks, such as reading e-mail and browsing web pages, must be condensed into smaller displays

• Some handheld devices use wireless technology, such as BlueTooth or 802.11, allowing remote access to e-mail and web browsing
• the limitations in the functionality of PDAs are balanced by their convenience and portability.

1.12 Computing Environments

1) Traditional Computing
2) Client-Server Computing
3) Peer-to-Peer Computing
4) Web-Based Computing

1.12.1 Traditional Computing

As computing matures, the lines separating many of the traditional computing environments are blurring.
• Used in office environment:
-PCs connected to a network, with servers providing file and print services.
• Used in home networks:
-At home, most users had a single computer with a slow modem.
- Today, network-connection speeds once available only at great cost are relatively inexpensive, giving home users more access to more data.
- Some homes have firewalls to protect their networks from security breaches. firewalls cost thousands of dollars a few years ago and did not even exist a decade ago.
• Web technologies are stretching the boundaries of traditional computing.
- Companies establish portals, which provide web accessibility to their internal servers.
- Network computers are terminals that understand web computing.
- Handheld PDAs can connect to wireless networks to use company's web portal.

• Systems were either batch or interactive.

- Batch system processed jobs in bulk, with predetermined input.
- Interactive systems waited for input from users.
-Time-sharing systems used a timer and scheduling algorithms to rapidly cycle processes through the CPU, giving each user a share of the resources

• Today, traditional time-sharing systems are uncommon

• The same scheduling technique is still in use on workstations and servers

1.12.2 Client-Server Computing

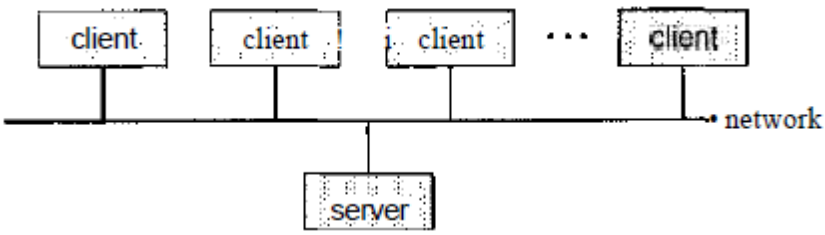
• As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture.

• Terminals connected to centralized systems are now being supplanted by PCs.

• Correspondingly, userinterface functionality once handled directly by the centralized systems is increasingly being handled by the PCs.

• As a result, many of todays systems act as server systems to satisfy requests generated by client systems.

• This form of specialized distributed system, called client-server system, has the general structure depicted in Figure below.



• Servers can be broadly categorized as : 1) Compute servers and 2) File servers

1) Compute-server system provides an interface to which a client can send a request to perform an action (for example, read data).

-In response, the server executes the action and sends back results to the client.

2) File-server system provides a file-system interface where clients can create, read, and delete files.

- For example: web server that delivers files to clients running web browsers.

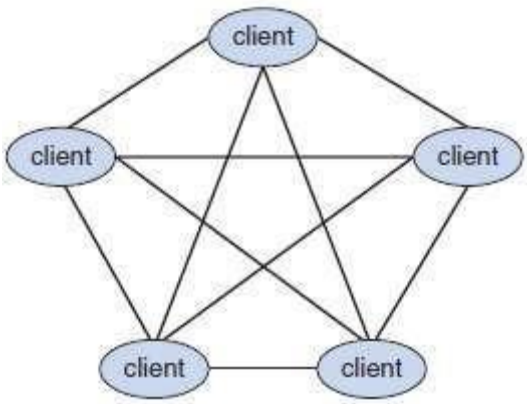
1.12.3 Peer-to-Peer Computing

• clients and servers are not distinguished from one another;

• all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service.

• Advantage:

- In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.



• To participate in a peer-to-peer system, a node must first join the network of peers.
• Once a node has joined the network, it can begin providing services to and requesting services from other nodes in the network.
• Determining what services are available is accomplished in one of two ways:

1) When a node joins a network, it registers its service with a centralized lookup service on the network.
-Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service.
Ex:Napster
2) A peer broadcasts a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer.
Ex:Gnutella

1.12.4 Web-Based Computing

•The Web has become ubiquitous, leading to more access by a wider variety of devices than was dreamt of a few years ago.
• PCs are still the most prevalent access devices, with workstations, handheld PDAs, and even cell phones also providing access.
• Web computing has increased the emphasis on networking.
• Devices that were not previously networked now include wired or wireless access.
• Devices that were networked now have faster network connectivity, provided by either improved networking technology, optimized network implementation code, or both.
• The implementation of web-based computing has given rise to new categories of devices, such as **load balancers,** which distribute network connections among a pool of similar servers.
• Generally, the Web has increased the complexity of devices, because their users require them to be web-enabled.

1.13 Operating System Services

• An OS provides an environment for the execution of programs.

• It provides services to

→ programs and

→ users of those programs

• Common functions helpful to the user are :

1) User Interface

-Almost all OS have a user-interface (UI).

-Different interfaces are:

i) CLI (Command Line Interface)

This uses

→ text commands and

→ method for entering the text commands.

ii) Batch Interface

-Commands & directives to control those commands are entered into files, and those files are executed.

iii) GUI (Graphical User Interface)

- The interface is a window-system with a pointing-device to

→ direct I/0

→ choose from menus

→ make selections.

→ key board to enter text

2) Program Execution

-The system must be able to

→ load a program into memory and → run the program.

- The program must be able to end its execution, either normally or abnormally.

3) I/O Operations

-a running program may require I/O(i.e file or i/o deivce)

-For specific devices, special functions may be desired (ex: to blank CRT screen).

- The OS must provide a means to do I/O operations because users cannot control I/O devices directly.

4) File-System Manipulation

-Programs need to

→ read & write files (or directories) → create & delete files

→ search for a given file and

→ allow or deny access to files.

5) Communications

-In some situations, one process needs to communicate with another process(may be on same system or on different systems connected on network)

-Communications may be implemented via

1. Shared memory or
2. Message passing

6) Error Detection

-Os must be constantly aware of possible errors

- Errors may occur in

→ CPU & memory-hardware (ex: power failure)

→ I/O devices (ex: lack of paper in the printer) and → user program (ex: arithmetic overflow)

-For each type of error, OS should take appropriate action to ensure correct & consistent computing.

- Debugging facilities can greatly enhance the user's and programmer's abilities to use the system efficiently

• Common functions for efficient operation of the system are:

1) **Resource Allocation**

-When multiple users are logged on the system at the same time, resources must be allocated to each of them.

-The OS manages different types of resources.

- Some resources (say CPU cycles) may have special allocation code. (CPU scheduling)

-Other resources (say I/O devices) may have general request & release code. (allocation of prinyers,modems etc)

2) Accounting

-We want to keep track of

→ which users use how many resources and

→ which kinds of resources.

- This record keeping may be used for

→ accounting (so that users can be billed) or

→ gathering usage-statistics.

Usage statistics may be a valuable tool for researchers who wish to reconfigiire the system to improve computing services

3) Protection and security

-The owners of information stored in a multiuser or networked computer system may want to control use of that information.

- When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the OS itself.

- Protection involves ensuring that all access to resources is controlled.

- Security starts with each user having authenticated to the system by means of a password.

- It extends to defending external I/O devices, including modems and network adapters, from invalid access attempts and to recording all such connections for detection of break-ins.

1.14 User Operating System Interface

• Two ways that users interface with the OS: 1) Command Interpreter (Command-line interface)
2) Graphical User Interface (GUI)

1) Command Interpreter

Some operating systems include the command interpreter in the kernel. Others, treat the command interpreter as a special program

• Main function:

To get and execute the next user-specified command.

• The commands are used to manipulate files i.e. create, copy, print, execute, etc.

• Two general ways to implement:

1) Command interpreter itself contains code to execute command.
-When a command is given the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call.
-In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.
2) Commands are implemented through system programs. This is used by UNIX.
-In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed.

Advantage:

-programmers can add new commands to the system easily by creating new files with the proper names.

-The command-interpreter program, which can be small, does not have to be changed for new commands to be added.

2) Graphical User Interfaces

• No entering of commands but the use of a mouse-based window and menu system
• The mouse is used to move a pointer to the position of an icon that represents
→ file
→ program
or
→ folder
• By clicking on the icon, the program is invoked.

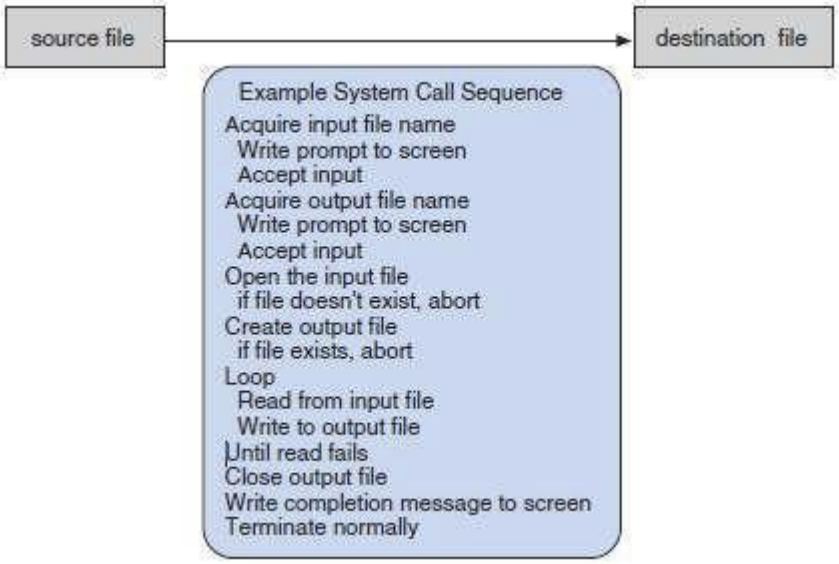
- The choice of whether to use a command-line or GUI interface is mostly one of personal preference.
- The user interface can vary from system to system and even from user to user within a system.
- The design of a useful and friendly user interface is therefore not a direct function of the operating system

1.15 System Calls

• These provide an interface to the OS services.

• These are available as routines written in C and C++.

•Figure illustrate system calls used for writing a simple program to read data from one file and copy them to another file.



•Most programmers never see this level of detail, however.

•Application developers design programs according to an **application programming interface (API).**

• The API

→ defines a set of functions that are available to the programmer

→ includes the parameters passed to functions and the return values.

•Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

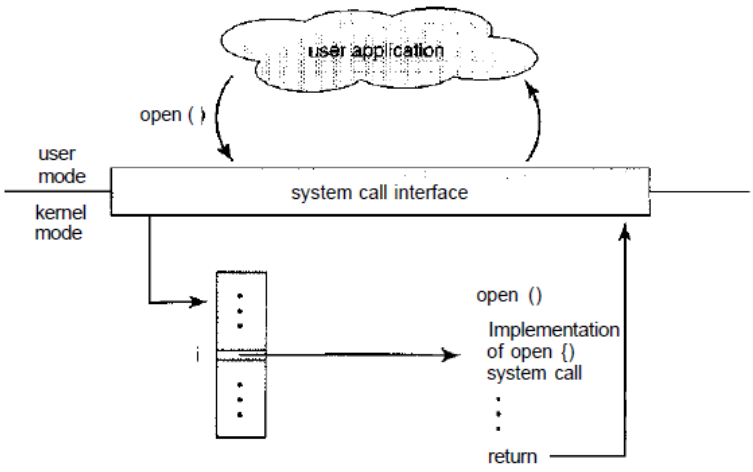
•The run-time support system (a set of functions built into libraries included with a compiler) for most programming languages provides a **system-call interface.**

•**system-call interface** serves as the link to system calls made available by the operating system.

•The system-call interface intercepts function calls in the API and invokes the necessary system call within the operating system.

•Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.

•The system call interface then invokes the intended system call in the operating system kernel and returns the status of the system call and any return values.



•The relationship between an API, the system-call interface, and the operating system is shown in Figure, which illustrates how the operating system handles a user application invoking the open() system call.

• Benefits of API:

1) Program portability.

2) Actual system-calls are more detailed (and difficult) to work with than the API available to the programmer.

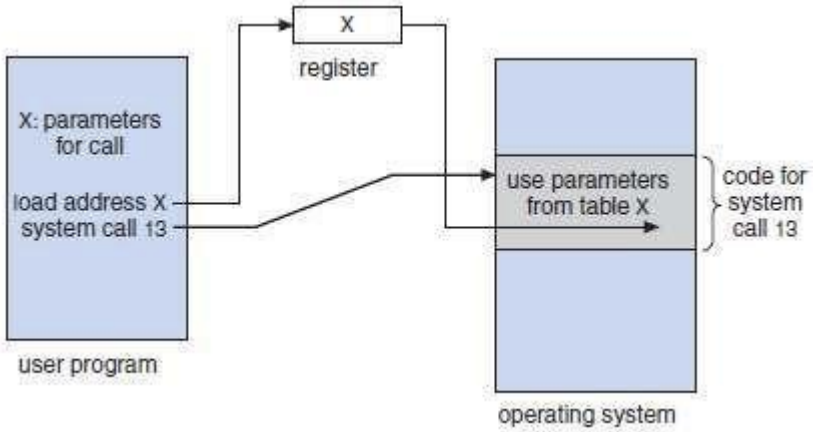
• System calls occur in different ways, depending on the computer in use.

• Three general methods are used to pass parameters to the operating system.

• Three general methods are used to pass parameters to the OS:

1) via registers.

2) Using a table in memory & the address is passed as a parameter in a register (as illustrated in Figure below).



3) The use of a stack is also possible where parameters are pushed onto a stack and popped off the stack by the OS.

• Some operating systems prefer the block or stack method, because those approaches do not limit the number or length of parameters being passed

1.16 Types of System Calls

System calls can be grouped roughly into five major categories:

1) Process control
2) File management
3) Device management
4) Information maintenance
5) Communications

Figure below summarizes the types of system calls normally provided by an operating system.

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices
- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

1.16.1 Process Control

• System calls used:

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

• A running program needs to be able to halt its execution either normally (end) or abnormally (abort).

• If program runs into a problem, error message may be generated and dumped into a file.

-This file can be examined by a debugger to determine the cause of the problem.

• The OS must transfer control to the next invoking command interpreter.

- •Command interpreter then reads next command.
- •In interactive system, the command interpreter simply continues with next command.
- •In GUI system, a pop-up window will request action from user.

How to deal with new process?

• A process executing one program can load and execute another program.

• Where to return control when the loaded program terminates?

The answer depends on the existing program:

1) If control returns to the existing program when the new program terminates, we must save the memory image of the existing program. (Thus, we have effectively created a mechanism for one program to call another program).

2) If both programs continue concurrently, we created a new process to be multiprogrammed.

• We should be able to control the execution of a process. i.e. we should be able to determine and reset the attributes of a process based on:

→ job's priority or

→ maximum execution time

• We may also want to terminate process that we created if we find that it

→ is incorrect or

→ is no longer needed.

• We may need to wait for processes to finish their execution.

• We may want to wait for a specific event to occur.

• The processes should then signal when that event has occurred.

1.16.2 File Management

• System calls used:
- create file, delete file open, close
- read, write, reposition
- get file attributes, set file attributes

• Working procedure:
1) We need to create and delete files.
2) Once the file is created,
→ we need to open it and to use it.
→ we may also read or write.
3) Finally, we need to close the file.

• We need to be able to
→ determine the values of file-attributes and
→ reset the file-attributes if necessary.

• File attributes include
→ file name
→ file type
→ protection codes and
→ accounting information.

1.16.3 Device Management

• System calls used:
- request device, release device;
- read, write, reposition;
- get device attributes, set device attributes;
- logically attach or detach devices.

• A program may need several resources to execute such as
→ memory
→ tape drives or
→ files.

• If the resources are available, they can be granted, and control can be returned to the user program;
-If the resources are unavailable, the program may have to wait until sufficient resources are available.

• Files can be thought of as virtual devices. Thus, many of the system calls used for files are also used for devices.

• In multi-user environment,
-We must first request the device, to ensure exclusive use of it.
-After we are finished with the device, we must release it.

• Once the device has been requested (and allocated), we can read and write the device.

• Due to lot of similarity between I/O devices and files, OS (like UNIX) merges the two into a combined file-device structure.

• UNIX merges I/O devices and files into a combined file-device structure.

1.16.4 Information Maintenance

• System calls used:

-get time or date, set time or date

- get system data, set system data

-get process, file, or device attributes

- set process, file, or device attributes

• Many system calls exist simply for the purpose of transferring information between the user program and the OS.

For ex,

1) Most systems have a system call to return
   → current time and
   → current date.
2) Other system calls may return information about the system, such as
   → number of current users
   → version number of the OS
   → amount of free memory or disk space.
3) The OS keeps information about all its processes, and there are system calls to access this information.

1.16.5 Communication

• System calls used:

-create, delete communication connection send, receive messages

- transfer status information

- attach or detach remote devices

• Two models of communication.

1) Message-passing model and 2) Shared Memory Model

1.16.5.1 Message Passing Model

• Information is exchanged through an IPC(inter process communication) provided by OS.

• Steps for communication:

1) Firstly, a connection must be opened using open connection system-call.

2) Each computer has a host-name, such as an IP name.
   -Similarly, each process has a process-name, which is translated into an equivalent identifier.
   -The **get hostid & get processid** system-calls do this translation.

3) Then, identifiers are passed to the **open and close** or **open connection and close connection** system-calls.

4) The recipient-process must give its permission for communication to take place with an accept connection system-call.

5) The processes that will be receiving connections are called daemons processes which,
   → execute a wait for connection system-call and
   → are awakened when a connection is made.

6) Then, client & server exchange messages by **read message** and **write message** system calls.

7) Finally, **the close connection** system-call terminates the communication.

• Advantages:

1) Useful when smaller numbers of data need to be exchanged.

2) It is also easier to implement than is shared memory.

1.16.5.2 Shared Memory Model

• Processes use shared memory create and shared memory attach system-calls to gain access to regions of memory owned by other processes.

• normally, the operating system tries to prevent one process from accessing another process's memory.

• Shared memory requires that two or more processes agree to remove this restriction

• Several processes exchange information by reading and writing data in the shared memory.

• The shared memory

→ is determined by the processes and

→ are not under the control of OS.

• The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

• Advantage:

1) Shared memory allows maximum speed(because it works at memory speed) and convenience of communication,

• Disadvantage:

1) Problems exist in the areas of protection and synchronization.

1.17 System Programs

•They provide a convenient environment for program development and execution.

• Some of them are simply user interfaces to system calls; others are considerably more complex.

• They can be divided into these categories:

1) File Management

- These programs create, delete, copy, print ,list and rename files and manipulate files and directories

2) Status Information

-Some programs ask the system for

- date (or time)
- available amount of memory(or disk space) or
- no. of users.

-Others are more complex, providing detailed performance, logging, and debugging information

-These information is formatted and then printed to the terminal (or output-device or file).

-Some systems also support a registry, which is used to store and retrieve configuration information

3) File Modification

-Text editors can be used to create and modify the content of files stored on disk or other storage devices.

4) Programming Language Support

-Compilers, assemblers, and interpreters for common programming-languages (such as C, C++) are provided to the user with the OS.a

5) Program Loading & Execution

Once a program is assembled or compiled, it must be loaded into memory to be executed

-The system may provide

→ absolute loaders
→relocatable loaders
→linkage editors and
→ overlay loaders.

-Debugging-systems are also needed.

6) Communications

-These programs are used for creating virtual connections between

→ processes
→ users and
→ computer-systems.

-They allow users to

→ browse web-pages
→ send email or
→ log-in remotely.

- Most OSs are supplied with programs that

→ solve common problems or
→ perform common operations.

-Such programs include

→ web-browsers , word-processors , spreadsheets and games.

-These programs are known as **system utilities** or application programs.

- The view of the operating system seen by most users is defined, by the application and system programs, rather than by the actual system calls.

1.18 Operating System Design & Implementation

1.18.1 Design Goals

• The first problem in designing a system is to
→ define goals and
→ define specifications.
• At the highest level the design of the system will be affected by
→ choice of hardware and
→ type of system such as
- batch ,time shared, single user, multiuser distributed,real time or general purpose
• Two basic groups of requirements: 1) User goals and 2) System goals

1) User Goals

Users desire certain obvious properties in a system:
→ convenient to use
→ easy to learn and to use
→ reliable, safe, and fast.
-these specifications are not particularly useful in the system design, since there is no general agreement on how to achieve them.

2) System Goals

-set of requirements can be defined by those people who must design, create, maintain, and operate the system:
→ easy to design
→ implement, and maintain
→ flexible, reliable, error free, and efficient.
-these requirements are vague and may be interpreted in various ways.
-There is, in short, no unique solution to the problem of defining the requirements for an operating system
-Specifying and designing an operating system is a highly creative task.

1.18.2 Mechanisms & Policies

• Mechanisms determine how to do something.
• For example, the timer construct is a mechanism for ensuring CPU protection, but deciding how long the timer is to be set for a particular user is a policy decision.
• Policies determine what will be done.
• Separating policy and mechanism is important for flexibility.
• Policies change over time; mechanisms should be general
-A change in policy would then require redefinition of only certain parameters of the system.

1.18.3 Implementation

• Traditionally, operating systems have been written in assembly language.
• OS's are nowadays written in higher-level languages like C/C++
• Advantages of higher-level languages:
1) Faster development and
2) OS is easier to port.

For example, MS-DOS was wrftten in Intel 8088 assembly language. Consequently, it is available on only the Intel family of CPUs. The Linux operating system, in contrast, is written mostly in C and is available on a number of different CPUs, including Intel 80X86, Motorola 680X0, SPARC, and MIPS RXOO0.

• Disadvantages of higher-level languages:

1) Reduced speed and

2) Increased storage requirements.

-But this is no longer a major issue in today's systems.

-A modern compiler can perform complex analysis and apply sophisticated optimizations that produce excellent code.

-Modern processors have deep pipelining and multiple functional units that can handle complex dependencies

• Although operating systems are large, only a small amount of the code is critical to high performance;

-After the system is written and is working correctly, bottleneck routines can be identified and can be replaced with assembly-language equivalents.

-To identify bottlenecks, we must be able to monitor system performance.

-Code must be added to compute and display measures of system behavior.

Example: trace listings of system behavior

-All interesting events are logged with their time and important parameters and are written to a file.

-Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies.

1.19 Operating System Structure

-based on how diffrent components of OS are arranged

1) Simple Structure
2) Layered Approach
3) Micro-kernels
4) Modules

1.19.1 Simple Structure

• These OSs are small, simple, and limited system.

• For example: MS-DOS and UNIX

1) MS-DOS was written to provide the most functionality in the least space.

-Figure shows its structure

- application programs are able to access the basic I/O routines

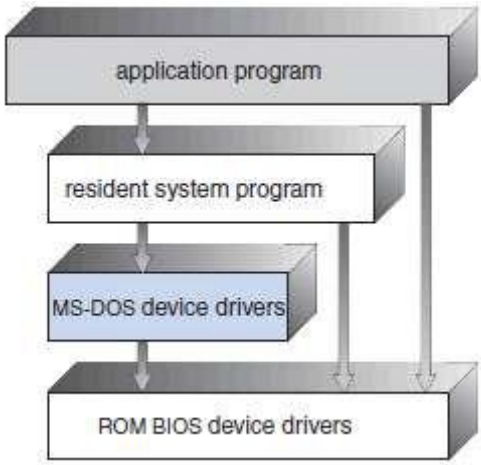
to write directly to the display and disk drives.

- Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.
- MS-DOS was also

-limited by the hardware of its era.

-no dual mode and

-no hardware protection,

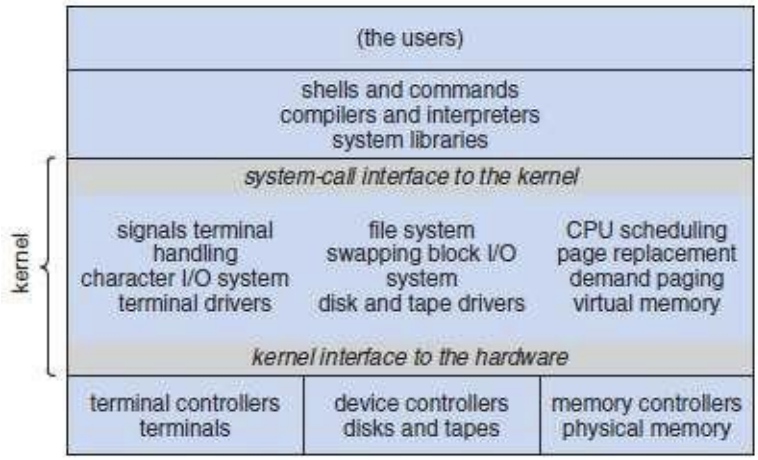


MS-DOS layer structure

- the designers of MS-DOS had no choice but to leave the base hardware accessible.

2) UNIX was initially limited by hardware functionality.

- Two separable parts of UNIX : 1) Kernel and
  2) System programs
- The kernel is further separated into a series of interfaces and device drivers as shown in figure below
- Everything below the system-call interface and above the physical hardware is the kernel.
- The kernel provides following funtions through system calls:
  → file system
  → CPU scheduling and
  → memory management.

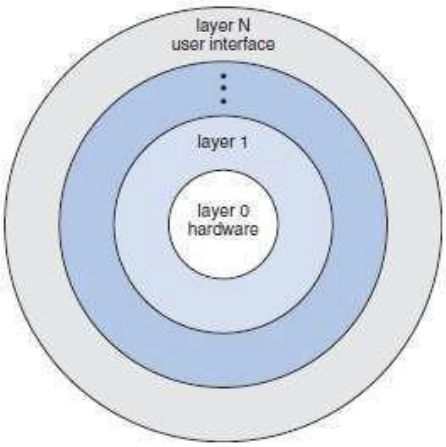


- Disadvantage:

1) Difficult to enhance, as changes in one section badly affects other areas.
2) This monolithic structure was difficult to implement and maintain.

1.19.2 Layered Approach

- A system can be made modular in many ways.
- One method is the **layered approach,** in which the operating system is broken up into a number of layers
- (levels).
- The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- This layering structure is depicted in Figure.
- An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data.
- A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer *M,* in turn, can invoke operations on lower-level layers.



- **Advantage:**
  - The main advantage of the layered approach is simplicity of construction and debugging.

-If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged.
-Thus, the design and implementation of the system is simplified.
-Each layer is implemented with only those operations provided by lowerlevel layers.
-A layer does not need to know how these operations are implemented;
it needs to know only what these operations do.
-Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

Problems:

- The major difficulty with the layered approach involves appropriately defining the various layers.

-Because a layer can use only lower-level layers, careful planning is necessary.

- A final problem with layered implementations is that they tend to be less efficient than other types.

At each layer, the parameters may be modified, data may need to be passed, and so on.
Each layer adds overhead to the system call; the net result is a system call that takes longer than does one on a nonlayered system.

1.19.3 Microkernels

• as UNIX expanded, the kernel became large and difficult to manage.
• Solution : the **microkernel** approach.
• This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs.
• The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.
• Communication is provided by *message passing*
For example, if the client program wishes to access a file, it must interact with the file server.
The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.

Benefit:

- ease of extending the operating system.
- All new services are added to user space and consequently do not require modification of the kernel.
- When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since
- most services are running as user—rather than kernel—processes.
- If a service fails, the rest of the operating system remains untouched.

Examples:

1) Tru64 UNIX - a UNIX interface to the user, but it is implemented with a Mach kernel.
   -The Mach kernel maps UNIX system calls into messages to the appropriate user-level services.
2) QNX: a real-time operating system that is also based on the microkernel design. The QNX microkernel provides services for message passing and process scheduling. It also handles low-level network

Disadvantages:

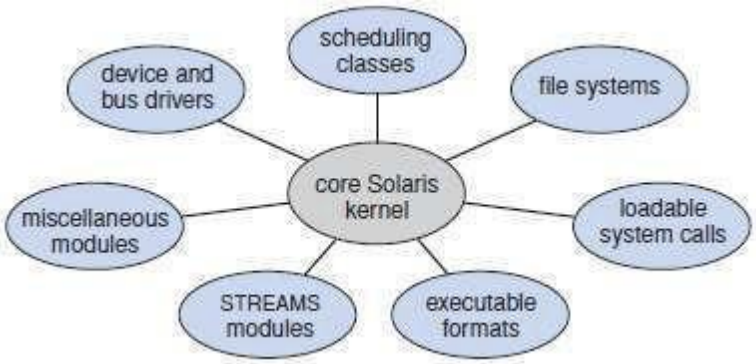
-microkernels can suffer from performance decreases due to increased system function overhead

1.19.4 Modules

- Perhaps the best current methodology for operating-system design involves using object-oriented programming techniques to create a modular kernel.
- the kernel has a set of core components and dynamically links in additional services either during boot time or during run time.
- Such a strategy uses dynamically loadable modules and is common in modern implementations of UNIX, such as Solaris, Linux, and Mac OS X.

- For example, the Solaris operating system structure, as shown in Figure, is organized around a core kernel with seven types of loadable kernel modules:

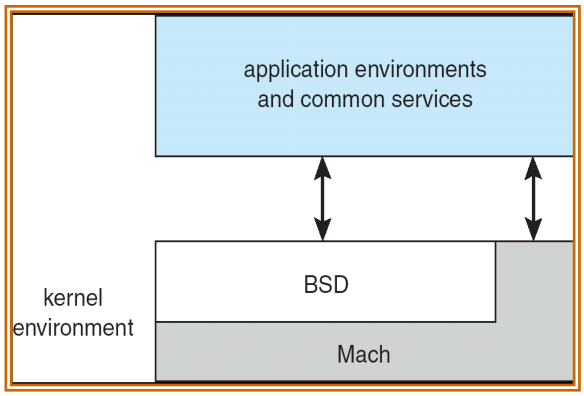
1. Scheduling classes
2. File systems
3. Loadable system calls
4. Executable formats
5. STREAMS modules
6. Miscellaneous
7. Device and bus drivers



- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module.
- Furthermore, the approach is like the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

- **The Apple Macintosh Mac OS X** operating system uses a **hybrid structure**.
- Mac OS X (also known as *Darwin)* structures the operating system using a layered technique where one layer consists of the Mach microkernel as shown in figure below.

• The top layers include
  → application environments and
  → set of services providing a graphical interface to applications.
• Kernel environment consists primarily of
  → Mach microkernel and
  → BSD kernel.
• Mach provides
  → memory management;
  → support for RPCs(remote procedure call) & IPC
  → thread scheduling.
• BSD component provides
  → BSD command line interface
  → support for networking and file systems
  → implementation of POSIX APIs
• The kernel environment provides an I/O kit for development of
  → device drivers and
  → dynamic loadable modules (which Mac OS X refers to as kernel extensions).



- As shown in the figure, applications and common services can make use of either the Mach or BSD facilities directly.

1.20 Virtual Machines

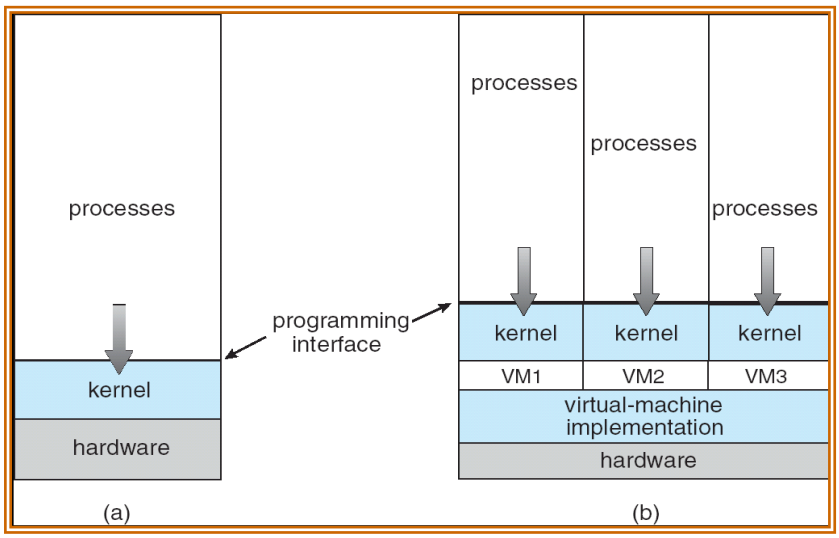
The fundamental idea:

To abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate execution environment is running its own private computer.

By using CPU scheduling and virtual-memory techniques , an operating system can create the illusion that a process has its own processor with its own (virtual) memory.

The virtual-machine provides an interface that is *identical* to the underlying bare hardware.

Each process is provided with a (virtual) copy of the underlying computer as shown in figure below.



Challenge:

-A major difficulty with the virtual-machine approach involves disk systems.

-the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling.

The solution

-is to provide virtual disks—termed *minidisks* in IBM's VM operating system —that are identical in all respects except size.

-the sum of the sizes of all minidisks must be smaller than the size of the physical disk space available.

-Users thus are given their own virtual machines.

-They can then run any of the operating systems or software packages that are available on the underlying machine.

1.20.1 Implementation

- Although the virtual-machine concept is useful, it is difficult to implement.
- Much work is required to provide an *exact* duplicate of the underlying machine.
- We know that the underlying machine has two modes: user mode and kernel mode.
- The virtual-machine software can run in kernel mode, since it is the operating system.
- The virtual machine itself can execute in only user mode.
- Just as the physical machine has two modes, the virtual machine also must have two modes.
- i.e, virtual user mode and a virtual kernel mode, both of which run in a physical user mode.
- Those actions that cause a transfer from user mode to kernel mode on a real machine (such as a system call or an attempt to execute a privileged instruction) must also cause a transfer from virtual user mode to virtual kernel mode on a virtual machine.

- Such a transfer can be accomplished as follows.

-When a system call, for example, is made by a program running on a virtual machine in virtual user mode, it will cause a transfer to the virtual-machine monitor in the real machine

-When the virtual-machine monitor gains control, it can change the register contents and program counter for the virtual machine to simulate the effect of the system call to show that it is now invirtual kernel mode.

1.20.2 Benefits

The virtual-machine concept has several advantages.

-there is complete protection of the various system resources.

-Each virtual machine is completely isolated from all other virtual machines, so there are no protection problems.

-however, there is no direct sharing of resources.

-Two approaches to provide sharing have been implemented.

1)it is possible to share a minidisk and thus to share files.

-This scheme is modeled after a physical shared disk but is implemented by software.

2) it is possible to define a network of virtual machines, each of which can send information over the virtual communications network.

-Again, the network is modeled after physical communication networks but is implemented in software.

-virtual-machine system is a perfect vehicle for operating-systems research and development. Normally, changing an operating system is a difficult task.

-The operating system, however, runs on and controls the entire machine.

-Therefore, the current system must be stopped and taken out of use while changes are made and tested.

-This period is commonly called *system development time.*

-Since it makes the system unavailable to users, system development time is often scheduled late at night or on weekends, when system load is low.

-A virtual-machine system can eliminate much of this problem.

-System programmers are given their own virtual machine, and system development is done on the virtual machine instead of on a physical machine.

1.20.3 Examples

1) VMware

- VMware is a popular commercial application that abstracts Intel 80X86 hardware into isolated virtual machines.
- VMware runs as an application on a host operating system such as Windows or Linux and allows this host system to concurrently run several different **guest operating systems** as independent virtual machines.

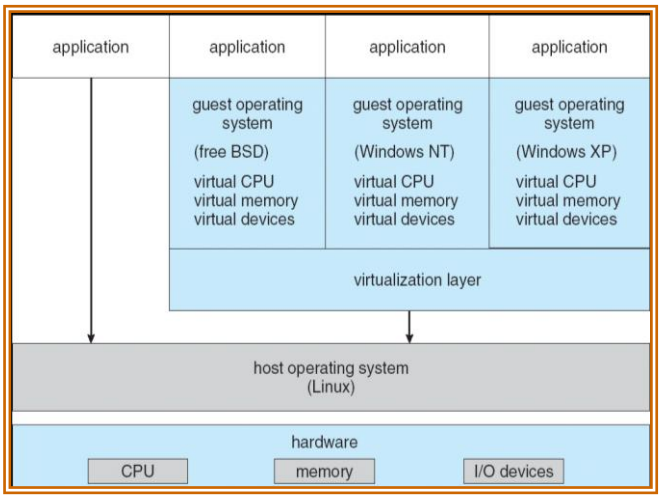
- Consider the following scenario:

  - A developer has designed an application and would like to test it on Linux, FreeBSD, Windows NT, and Windows XP.

  -Such testing could be accomplished *concurrently* on the same physical computer using VMware.

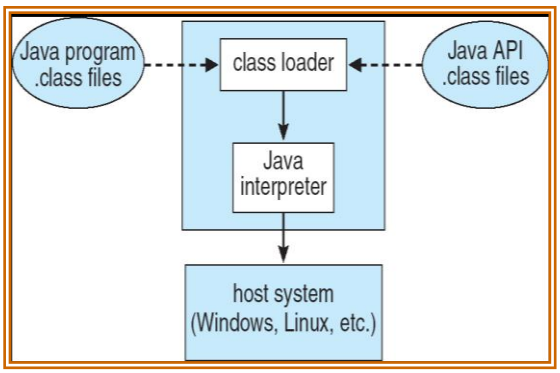
  -In this case, the programmer could test the application on a host operating system and on three guest operating systems with each system running as a separate virtual machine as shown in figure.

In this scenario,
-Linux is running as the host operating system;
-FreeBSD, Windows NT, and Windows XP are running as guest operating systems.
-The virtualization layer is the heart of VMware, as it abstracts the physical hardware into isolated virtual machines running as guest operating systems.
-Each virtual machine has its own virtual CPU, memory, disk drives, network interfaces, and so forth.



2) The Java Virtual Machine

- Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995.
- In addition to a language specification and a large API library, Java also provides a specification for a Java virtual machine—or JVM.
- For each Java class, the compiler produces an architecture-neutral **bytecode** output (.class) file that will run on any implementation of the JVM.
- The JVM is a specification for an abstract computer.
- It consists of a **class loader** and a Java interpreter that executes the architecture-neutral bytecodes, as shown in Figure .
- The class loader loads the compiled . class files from both the Java program and the Java API for execution by the Java interpreter.
- After a class is loaded, the verifier checks that the . class file is valid Java bytecode .
- If the class passes verification, it is run by the Java interpreter.



- The JVM also automatically manages memory by performing **garbage collection**—the practice of reclaiming memory from objects no longer in use and returning it to the system.
- The JVM may be implemented in software on top of a host operating system, such as Windows, Linux, or Mac OS X, or as part of a web browser.
- Alternatively, the JVM may be implemented in hardware on a chip specifically designed to run Java programs.

1.21 Operating System Generation

OS is designed to run on any of a class of machines.

the system must be configured or generated for each specific computer site

SYSGEN (system generation) is a process used for configuring a system for each specific computer site

The SYSGEN program reads from a given file, or asks the operator of the system for information concerning the specific configuration of the hardware system, or probes the hardware directly to determine what components are there.

The following kinds of information must be determined:

• SYSGEN program must determine:

1) What CPU will be used?
2) How will boot disk be formatted?
3) How much memory is available?
4) What devices are available?
5) What OS options are desired what parameter values are to be used?

Once this information is determined, it can be used in several ways:

i) a system administrator can use it to modify a copy of the source code of the operating system.
-The operating system then is completely compiled to output object version of the operating system.

ii) the system description can cause the creation of tables and the selection of modules from a precompiled library.
-These modules are linked together to form the generated operating system.
-Because, the system is not recompiled, system generation is faster, but the resulting system may be overly general.

iii) it is possible to construct a system that is completely table driven.
-All the code is always part of the system, and selection occurs at execution time, rather than at compile or link time.
-System generation involves simply creating the appropriate tables to describe the system.

1.22 System Boot

- After an operating system is generated, it must be made available for use by the hardware.
- The procedure of starting a computer by loading the kernel is known as *booting* the system.
- a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution.
- Some computer systems, such as PCs, use a two-step process
  i) a simple bootstrap loader fetches a more complex boot program from disk,
  ii) which in turn loads the kernel.

• When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location that is location of initial bootstrap program and execution starts there.

• This program is in the form of **read-only memory (ROM),** because
-the RAM is in an unknown state at system startup.
-ROM is convenient because it needs no initialization and cannot be infected by a computer virus.

- Some systems—such as cellular phones, PDAs, and game consoles—store the entire operating system in ROM as they are small operating systems,

Problem:

-changing the bootstrap code requires changing the ROM hardware chips.

-Some systems resolve this problem by using **erasable programmable read-only memory** (EPROM),

-All forms of ROM are also known as **firmware,** since their characteristics fall somewhere between those of hardware and those of software.

-A problem with firmware in general is that executing code there is slower than executing code in RAM and relatively expensive, so usually only small amounts are available.

- For large operating systems the bootstrap loader is stored in firmware, and the operating system is on disk.

-the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block.**

-the program stored in the boot block may load the entire operating system into memory and begin its execution.

-A disk that has a boot partition called a **boot disk** or system disk.

- Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution.
- It is only at this point that the system is said to be **running.**

MODULE 2
PROCESS MANAGEMENT

2.1. Process Concept

• A process is the unit-of-work in a modern time-sharing system.

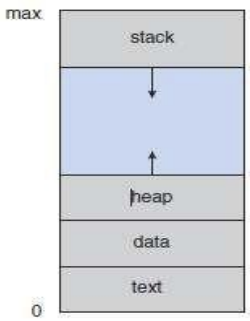
• A system consists of a collection of processes:

- OS process can execute system-code and
- User process can execute user-code.

2.1.1 The Process

• A process is a program in execution.

• A process is more than the program code, which is sometimes known as the **text section.**



• It also includes :

- **Program Counter** to indicate the address of next instruction to be executed.
- **Registers Content** of the processor.
- **Process Stack** contains temporary data.
- **Data Section** contains global variables.
- **Heap** is memory that is dynamically allocated during process run time.

• The structure of a process in memory is as shown in Figure.

• A program by itself is not a process.

1) A program is a passive-entity such as an executable-file stored on disk.

2) A process is an active-entity with a program counter specifying the next instruction to execute and a set of associated resources

• A program becomes a process when an executable-file is loaded into memory.

• If you run many copies of a program, each is a separate process.

• The text-sections are equivalent, but the data-sections vary.

2.1.2 Process State

• As a process executes, it changes state.

•The state of a process is defined in part by the current activity of that process.

• Each process may be in one of the following states (as shown in Figure):

1) **New:** The process is being created.

2) **Running:** Instructions are being executed.

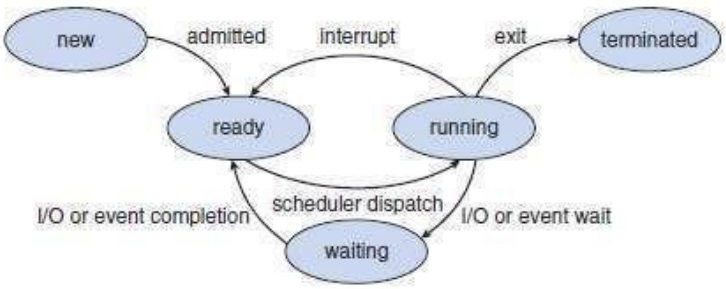
3) **Waiting:** The process is waiting for some event to occur (such as I/0 completions).

4) **Ready:** The process is waiting to be assigned to a processor.

5) **Terminated:** The process has finished execution.

• Only one process can be running on any processor at any instant.

•Many processes may be *ready* and *limiting*, however.



2.1.3 Process Control Block

• Each process in OS is represented by a PCB (Process Control Block) —also called a *task control block.*

• PCB contains following information about the process:

1) Process State

-The current state of process may be

→ new

→ ready

→ running

→ waiting or

→ halted.

2) Program Counter

-This indicates the address of the next instruction to be executed for the process.

3) CPU Registers

- These include

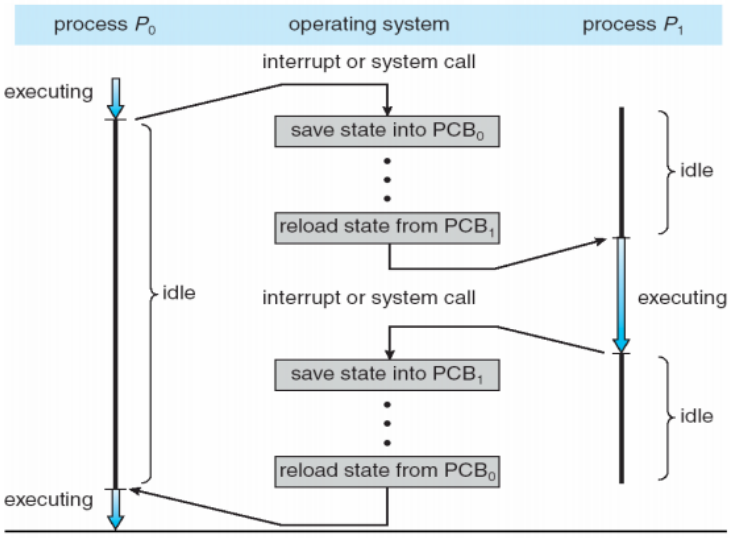
→ accumulators (AX)

→ index registers (SI, DI)

→ stack pointers (SP) and

→ general-purpose registers (BX, CX, DX).

Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward as shown in figure below



4) CPU Scheduling Information

-This includes

→ priority of process

→ pointers to scheduling-queues and

→ scheduling-parameters.

5) Memory Management Information

- This includes

→ value of base- & limit-registers and

→ value of page-tables(or segment-tables).

6) Accounting Information

- This includes

→ amount of CPU time

→ time-limit and

→ process-number.

7) I/O Status Information

- This includes

→ list of I/O devices allocated

→ list of open files.

In brief, the PCB simply serves as the repository for any information that may vary from process to process.

2.1.4 Threads

- The process model discussed so far has implied that a process is a program that performs a single **thread** of execution.
- For example,
  - -when a process is running a word-processor program, a single thread of instructions is being executed.
  - -This single thread of control allows the process to perform only one task at one time.
  - -For example,the user cannot simultaneously type in characters and run the spell checker within the same process,
- Many modern operating systems have extended the process concept to allow a process to have multiplethreads of execution and thus to perform more than one task at a time.

2.2 Process Scheduling

• Objective of multiprogramming:

To have some process running at all times to maximize CPU utilization.

• Objective of time-sharing:

To switch the CPU between processes so frequently that users can interact with each program while it is running.

• To meet above 2 objectives: Process scheduler is used to select an available process for program execution on the CPU.

•For a single-processor system, there will never be more than one running process.

•If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

2.2.1 Scheduling Queues

• types of scheduling-queues:

-Job Queue

- This consists of all processes in the system.
- As processes enter the system, they are put into a job-queue.

- Ready Queue

This consists of the processes that are

- residing in main-memory and
- ready & waiting to execute

-Device Queue

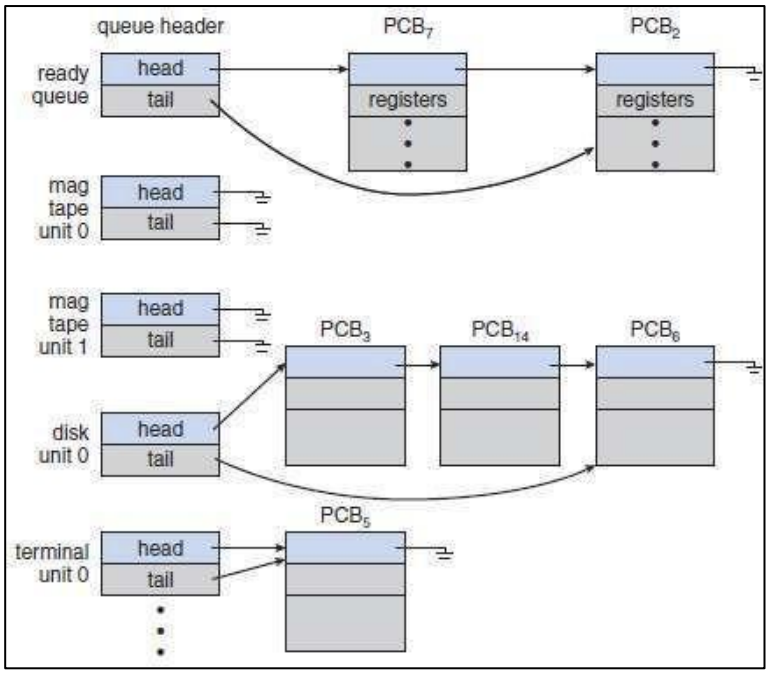
-When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request

-Since there are many processes in the system, the disk may be busy with the I/O request of some other process.

-The process therefore may have to wait for the disk and enters into Device Queue

- Each device has its own device-queue.

Figure shows The ready queue and various I/O device queues.



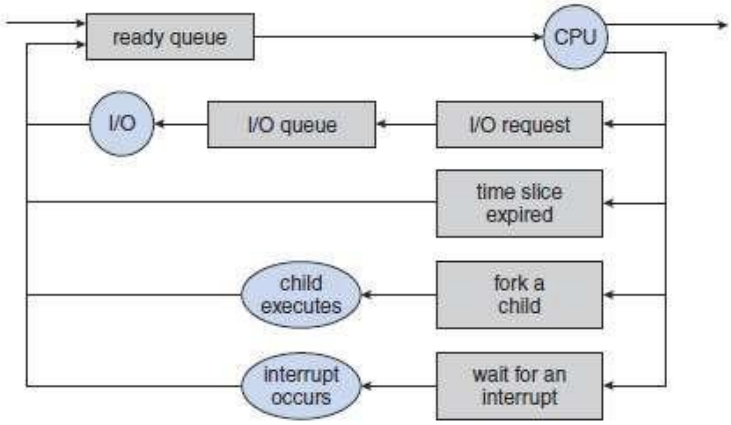
-These queues is generally stored as a linked list.

-A ready-queue header contains pointers to the first and final PCBs in the list.

-similarly a device queue header contains pointers to the first and final PCBs in the list.

-Each PCB has a pointer to the next PCB in the ready-queue.

• A common representation for a discussion of process scheduling is a **queueing diagram,** such as that in Figure below.



• Each rectangular box represents a queue.
• Two types of queues are present: the ready queue and a set of device queues.
• The circles represent the resources that serve the queues, and
• the arrows indicate the flow of processes in the system.
• A new process is initially put in the ready queue.
• It waits there till it is selected for execution, or is **dispatched.**
• Once the process is allocated the CPU and is executing, one of several events could occur:
  - The process could issue an I/O request and then be placed in an I/O queue.
  - The process could create a new subprocess and wait for the subprocess's termination.
  - The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.
• In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
• A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

2.2.2 Schedulers

• Three types of schedulers:

1) Long-term scheduler
2) Short-term scheduler and
3) Medium-term schedulers

Long-Term Scheduler	Short-Term Scheduler
Also called job scheduler.	Also called CPU scheduler.
Selects which processes should be brought into the ready-queue.	Selects which process should be executed next and allocates CPU.
Need to be invoked only when a process leaves the system and therefore executes much less frequently.	Need to be invoked to select a new process for the CPU and therefore executes much more frequently.
minutes may separate the creation of one new process and the next.	a process may execute for only a few milliseconds.
Controls the degree of multiprogramming.	

• Processes can be described as either:

- **I/O-bound Process**
  - Spends more time doing I/O operation than doing computations.
  -Many short CPU bursts.
- **CPU-bound Process**
  - Spends more time doing computations than doing I/O operation.
  -Few very long CPU bursts.

•long-term scheduler should select a **good process mix** of I/O-bound and CPU-bound processes

Because:

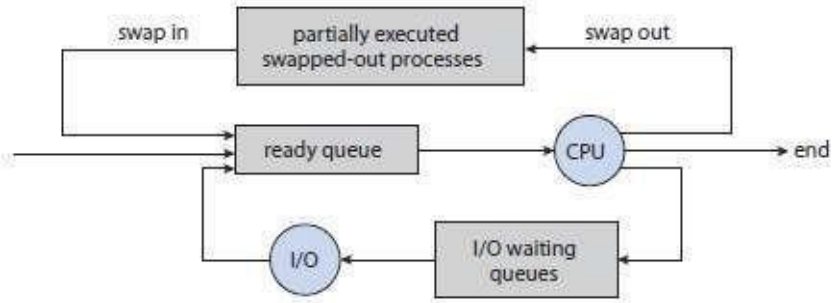
- If all processes are I/0 bound, then
  i) Ready-queue will almost always be empty,and
  ii) Short-term scheduler will have little to do.
- If all processes are CPU bound, then
  i) I/0 waiting queue will almost always be empty (devices will go unused) and
  ii) System will be unbalanced.

• Some time-sharing systems use **medium-term scheduler** as shown in figure below.

• As it is not possible to keep all process that are in ready queue in memory,hence
  - The scheduler removes processes from memory.
  - Later, the process can be reintroduced into memory, and its execution can be continued where it left off.

• This scheme is called swapping: The process is swapped out, and is later swapped in, by the scheduler.

• Thus reduces the degree of multiprogramming.



2.2.3 Context Switch

• Context-switch means saving the state of the old process and switching the CPU to another process.
• The context of a process is represented in the PCB of the process; it includes
→ value of CPU registers
→ process-state and
→ memory-management information.
• Disadvantages:
1) Context-switch time is pure overhead, because the system does no useful work while switching.
2) Context-switch times are highly dependent on hardware support.
For instance,
-some processor provide multiple sets of registers.
-A context switch here simply requires changing the pointer to the current register set.
-Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory,as before.
3) the more complex the operating system, the more work must be done during a context switch.
4) advanced memory-management techniques may require extra data to be switched with each context.
For instance, the address space of the current process must be preserved as the space of the next task is prepared for use.

2.3 Operations on Processes

-Process Creation and
- Process Termination

2.3.1 Process Creation

• A process may create a new process via a create-process system-call.
• The creating process is called a parent-process.
•The new process created by the parent is called the child-process (Sub-process).
• Each of these new processes may in turn create other processes, forming a **tree** of processes.
• OS identifies processes by unique process identifier (pid) ,which is typically an integer-number.
• A process needs following resources to accomplish the task:
→ CPU time
→ memory and
→ I/0 devices.
• Child-process may
→ get resources directly from the OS or
→ get resources of parent-process.
-The parent may have to partition its resources among its children, or
-parent may be able to share some resources (such as memory or files) among several of its children.
-Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.
• When a process creates a new process, two possibilities exist in terms of execution:
1) The parent & the children execute concurrently.
2) The parent waits until all the children have terminated.

•There are also two possibilities in terms of the address space of the new process:
1) The child-process is a duplicate of the parent-process (it has the same program and data as the parent).
2) The child-process has a new program loaded into it.

Process creation in UNIX

• In UNIX, each process is identified by its process identifier (pid), which is a unique integer.
• A new process is created by the fork() system-call
• The new process consists of a copy of the address-space of the original process.
• Both the parent and the child continue execution at the instruction after fork( ), but with one difference:
    1) The return value for the fork() is zero for the new (child) process.
    2) The return value for the fork() is nonzero, pid of the child for the parent-process.
• Typically, the exec() system-call is used after a fork() system-call by one of the two processes to replace the process's memory-space with a new program.
•The exec () system call loads a binary file into memory and starts its execution.
• The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait () system call to move itself off the ready queue until the termination of the child.
• The C program shown below illustrates the UNIX system calls previously described.

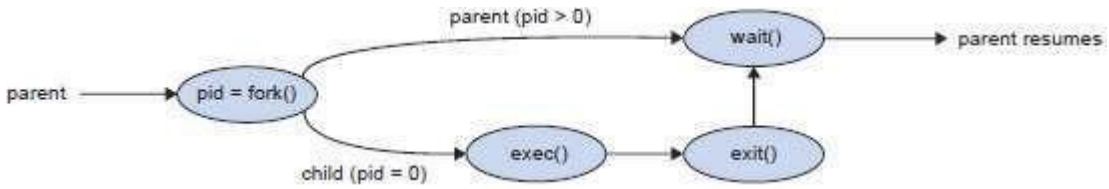
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

   /* fork a child process */
   pid = fork();                                  copy parent's space;
                                                  parent and child continue
   if (pid < 0) { /* error occurred */            executing in their own space */
      fprintf(stderr, "Fork Failed");
      return 1;
   }
   else if (pid == 0) { /* child process */
      execlp("/bin/ls","ls",NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
   }

   return 0;
}
```

- We now have two different processes running a copy of the same program.
- The child process overlays its address space with the UNIX command / bin/ls (used to get a directory listing) using the execlp( ) system call which is a version of the exec( ) system call).
- The parent waits for the child process to complete with the wait () system call.
- When the child process completes (by either implicitly or explicitly invoking exit( )) the parent process resumes from the call to wait (), where it completes using the exit( ) system call.
- This is also illustrated in Figure below.



2.3.2 Process Termination

• A process terminates when it executes the last statement (in the program).

• Then, the OS deletes the process by using exit( ) system-call.

• At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call).

• The wait () system call returns the process identifier of a terminated child so that the parent can tell which of its possibly many children has terminated.

• Then, the OS de-allocates all the resources of the process such as,

→ memory

→ open files and

→ I/0 buffers.

• Process termination can occur in following cases:

→ A process can cause the termination of another process

-for example using TerminateProcess() system-call

→ such a system call can be invoked only by the parent of the process that is to be terminated else users could arbitrarily kill the processes.

• A parent terminates the execution of children for following reasons:

- The child has exceeded its usage of some resources.
- The task assigned to the child is no longer required.
- The parent is exiting, and the OS does not allow a child to continue.

• In some systems, if a process terminates, then all its children must also be terminated. This phenomenon is referred to as **cascading termination**(normally initiated by OS)

• In UNIX ,if the parent terminates, however, all its children will be assigned to their new parent, **the init process.**

2.4. Inter Process Communication (IPC)

• Processes executing concurrently in the OS may be 1) Independent processes or
2) Co-operating processes.

1) A process is independent if

i) The process cannot affect or be affected by the other processes.

ii) The process does not share data with other processes.

2) A process is co-operating if

i) The process can affect or be affected by the other processes.

ii) The process shares data with other processes.

•There are several reasons for providing an environment that allows process cooperation(Advantages of process co-operation):

1) Information Sharing

- Since many users may be interested in same piece of information (ex: shared file).

2) Computation Speedup

- We must break the task into subtasks.
- Each subtask should be executed in parallel with the other subtasks.
- The speed can be improved only if computer has multiple processing elements such as
  → CPUs or
  → I/O channels.

3) Modularity

- Divide the system-functions into separate processes or threads.

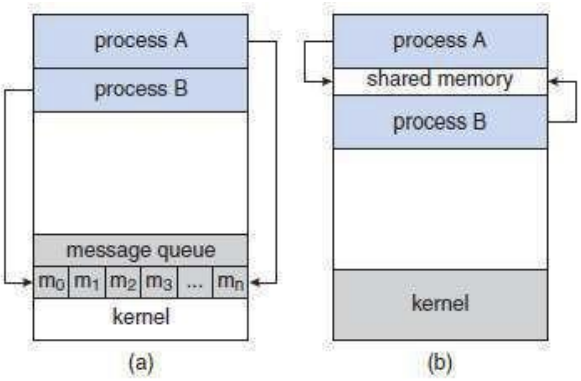
4) Convenience

- An individual user may work on many tasks at the same time.

For ex, a user may be editing, printing, and compiling in parallel.

• Two basic models of IPC 1) Shared-memory and
2) Message passing

The two communications models are contrasted in Figure below,



SHARED MEMORY	MESSAGE PASSING
a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region	In the messagepassing model, communication takes place by means of messages exchanged between the cooperating processes.
useful for exchanging lager amounts of data	useful for exchanging smaller amounts of data, because no conflicts need be avoided
Harder to implement since it requires synchronization	easier to implement
in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.	are typically implemented using system calls and thus require the more time consuming task of kernel intervention.
Relatively fast	Relatively slow
Figure (b)	Figure (a)

2.4.1 Shared-Memory Systems

• Communicating-processes must establish a region of shared-memory.

• A shared-memory resides in address-space of the process creating the shared-memory.

• Other processes that wish to communicate, must attach their address-space to the shared-memory.

• normally, the operating system tries to prevent one process from accessing another process's memory.

• Shared memory requires that two or more processes agree to remove this restriction.

• The processes can then exchange information by reading and writing data in the shared-memory.

• The form of the data and the location are determined by these processes and are not under the operating system's control.

• The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

• For ex, **Producer-Consumer Problem:**

- Producer-process produces information that is consumed by a consumer-process

Example: a compiler may produce assembly code, which is consumed by an assembler.

- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be
  - filled by the producer and
  - emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used.
  - The **unbounded buffer** places no practical limit on the size of the buffer.
    -The consumer may have to wait for new items, but the producer can always produce new items.
  - The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

- Usage of bounded buffer to enable processes to share memory.
- The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- The shared buffer is implemented as a circular array with two logical pointers: in and out.
- The variable
  - in points to the next free position in the buffer;
  - out points to the first full position in the buffer.
- The buffer is empty when in == out;
- the buffer is full when ((in + 1) % BUFFER_SIZE) == out.

The code for the producer and consumer processes is shown in Figures below.

```
item nextProduced;

while (true) {
     /* produce an item in nextProduced */
     while (((in + 1) % BUFFER-SIZE) == out)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
}
```

a) producer code

```
item nextConsumed;

while (true) {
     while (in == out)
        ; //do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

b)consumer code

- The producer process has a local variable nextProduced in which the new item to be produced is stored.
- The consumer process has a local variable nextConsumed in which the item to be consumed is stored.

2.4.2 Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- Useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- For example, a **chat** program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

- A message-passing facility provides at least two operations: **send(message) and receive(message).**
- Messages sent by a process can be of either fixed or variable size.
- If fixed-sized messages can be sent,
  - the system-level implementation is straightforward.
  - makes the task of programming more difficult.
- If variable-sized messages can be sent ,
  - a more complex system-level implementation,
  - but the programming task becomes simpler.

- If processes *P* and *Q* want to communicate, they must send messages to and receive messages from each other;
- A **communication link** must exist between them.
- Several methods for logically implementing a link and the send()/receive () operations:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

2.4.2.1. Naming

- Processes that want to communicate must have a way to refer to each other.
- They can use either direct or indirect communication.

- **direct communication,**

-each process that wants to communicate must explicitly name the recipient or sender of the communication.

-
- the send.0 and receive() primitives are defined as:
  - **send(P, message)** —Send a message to process P.
  - **receive (Q, message)** —Receive a message from process Q.

- A communication link in this scheme has the following properties:
  - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - A link is associated with exactly two processes.
  - Between each pair of processes, there exists exactly one link.

• ***symmetry* in addressing:**that is, both the sender process and the receiver process must name the other to communicate.

• ***asymmetry* in addressing**: only the sender names the recipient; the recipient is not required to name the sender.

• In this scheme, the send() and receive () primitives are defined as follows:

• **send(P, message**) —Send a message to process P.

• **receive(id, message) —-**Receive a message from any process; the variable *id* is set to the name of the process with which communication has taken place.

• Indirect communication,

the messages are sent to and received from mailboxes, or ports.

A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Each mailbox has a unique identification.

a process can communicate with some other process via a number of different mailboxes.

Two processes can communicate only if the processes have a shared mailbox, however.

The send( ) and receive ( ) primitives are defined as follows:

• send(A, message)—Send a message to mailbox A.

• receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

• A link is established between a pair of processes only if both members of the pair have a shared mailbox.

• A link may be associated with more than two processes.

• Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

- Now suppose that processes *P1, P2,* and P3 all share mailbox A .
- Process Pi sends a message to *A,* while both P2 and P3 execute a receive() from A
- Which process will receive the message sent by Pi?
- The answer depends on which of the following methods we choose:

• Allow a link to be associated with two processes at most.

• Allow at most one process at a time to execute a receive () operation.

• Allow the system to select arbitrarily which process will receive the message .

The system also may define an algorithm for selecting which process will receive the message

- A mailbox may be owned either by a process or by the operating system.

➢ **owned by a process**

If the mailbox is **owned by a process** (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox).

-When a process that owns a mailbox terminates, the mailbox disappears.

➢ **owned by the operating system**

-In contrast, a mailbox that is **owned by the operating system** has an existence of its own.

-It is independent and is not attached to any particular process.

-The operating system then must provide a mechanism that allows a process to do the following:

• Create a new mailbox.

• Send and receive messages through the mailbox.

• Delete a mailbox.

-The process that creates a new mailbox is that mailbox's owner by default.

-the ownership and receiving privilege may be passed to other processes through appropriate system calls.

-this provision could result in multiple receivers for each mailbox.

2.4.2.2 Synchronization

- Communication between processes takes place through calls to send( ) and receive( ) primitives.
- There are different design options for implementing each primitive.
- Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**
  - **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - **Nonblocking send.** The sending process sends the message and resumes operation.
  - **Blocking receive.** The receiver blocks until a message is available.
  - **Nonblocking receive.** The receiver retrieves either a valid message or a null.
- Different combinations of send( ) and receive( ) are possible.

2.4.2.3 Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
- Basically, such queues can be implemented in three ways:

• **Zero capacity.** The queue has a maximum length of zero;
-thus, the link cannot have any messages waiting in it.
-In this case, the sender must block until the recipient receives the message.

• **Bounded capacity.** The queue has finite length *n;* thus, at most *n* messages can reside in it.
-If the queue is not full when a new message is sent, the message is placed in the queue, and the sender can continue execution without waiting.
-The links capacity is finite, however.
-If the link is full, the sender must block until space is available in the queue.

• **Unbounded capacity.** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

- The zero-capacity case is sometimes referred to as a message system with no buffering;
- the other cases are referred to as systems with automatic buffering.

MULTI-THREADED PROGRAMMING

2.5 Multi-Threaded Programming

• A thread is a basic unit of CPU utilization.

• It consists of

→ thread ID

→ Program counter

→ register-set and

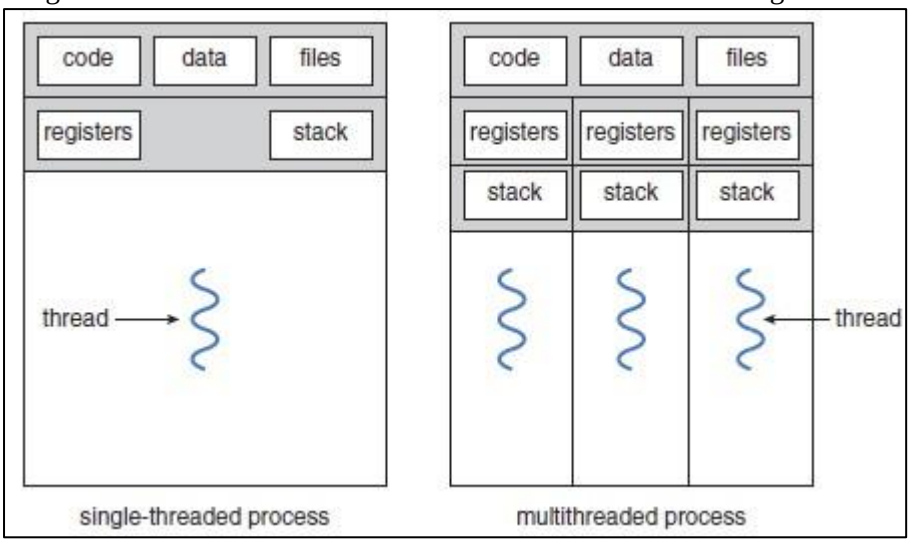
→ stack.

• It shares with other threads belonging to the same process its code-section & data-section.

• A traditional (or heavy weight) process has a single thread of control.

• If a process has multiple threads of control, it can perform more than one task at a time. Such a process is called multi-threaded process

• Figure below illustrates the difference between a traditional single-threaded process and a multithreaded process.



2.5.1 Motivation

1) The software-packages that run on modern PCs are multithreaded.

An application is implemented as a separate process with several threads of control.

For ex: A word processor may have

→ first thread for displaying graphics

→ second thread for responding to keystrokes and

→ third thread for performing grammar checking.

2) In some situations, a single application may be required to perform several similar tasks. For example,

- A web server accepts client requests for web pages, images, sound, and so forth.
- A busy web server may have several (perhaps thousands) of clients concurrently accessing it.
- If the web server ran as a traditional single-threaded process, it would be able to service only one
- client at a time.
- The amount of time that a client might have to wait for its request to be serviced could be enormous.

Solution:

- Let the server run as a single process that accepts requests.
- The server create a separate thread that would listen for client requests; when a request was made, rather than creating another process, the server would create another thread to service the request.
- Process creation is time consuming and resource intensive.
- It is generally more efficient to use one process that contains multiple threads.
- This approach would multithread the web-server process.

3)Threads also play a vital role in remote procedure call (RPC) systems

- RPC servers are multithreaded.
- When a server receives a message, it services the message using a separate thread.
- This allows the server to service several concurrent requests.

4) Most OS kernels are multithreaded;

- Several threads operate in kernel, and each thread performs a specific task, such as
  → managing devices or
  → interrupt handling.

2.5.2 Benefits

1) Responsiveness

• A program may be allowed to continue running even if part of it is blocked.

• Thus, increasing responsiveness to the user.

• For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.

2) Resource Sharing

• By default, threads share the memory (and resources) of the process to which they belong.

Thus, an application is allowed to have several different threads of activity within the same address-space.

3) Economy

- Allocating memory and resources for process-creation is costly.
- Thus, it is more economical to create and context-switch threads when compared to process.
- it is much more time consuming to create and manage processes than threads
- In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

4) Utilization of Multiprocessor Architectures

- The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.
- A singlethreaded process can only run on one CPU, no matter how many are available.
- Multithreading on a multi-CPU machine increases concurrency

2.6 Multi-Threading Models

• Support for threads may be provided at either

-The user level, for user threads or

- By the kernel, for kernel threads.

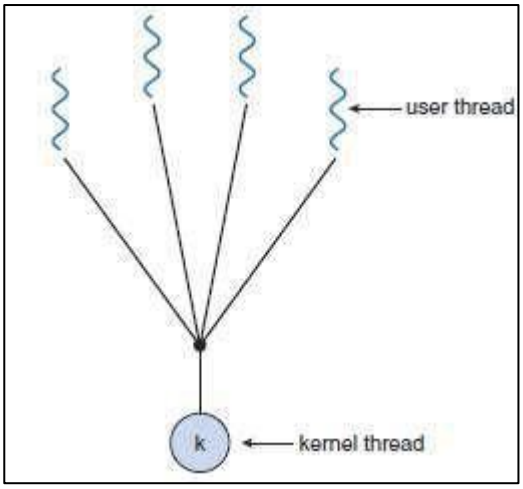
•User-threads are supported above the kernel and are managed without kernel support.

• Kernel-threads are supported and managed directly by the OS.

• Three ways of establishing relationship between user-threads & kernel-threads:

- Many-to-one model
- One-to-one model and
- Many-to-many model.

2.6.1 Many-to-One Model



• Many user-level threads are mapped to one kernel thread as shown in figure..

• Advantage:

1) Thread management is done by the thread library in user space, so it is efficient.

• Disadvantages:

1) The entire process will block if a thread makes a blocking system-call.

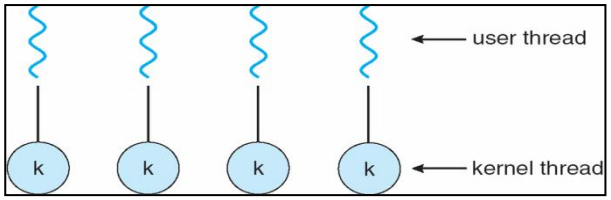
2) Multiple threads are unable to run in parallel on multiprocessors, as only one thread can access the kernel at a time.

• For example:

→ Solaris Green Threads

→ GNU Portable Threads.

2.6.2 One-to-One Model



• Each user thread is mapped to a kernel thread as shown in figure.

• Advantages:

1) It provides more concurrency by allowing another thread to run when a thread makes a blocking system-call.

2) Multiple threads can run in parallel on multiprocessors.

• Disadvantage:

1) Creating a user thread requires creating the corresponding kernel thread.

• For example:

→ Windows NT/XP/2000

→ Linux

2.6.3 Many-to-Many Model

• Many user-level threads are multiplexed to a smaller or equal number of kernel threads as shown in Figure

• The number of kernel threads may be specific to either a particular application or a particular machine

• **the many-to-one model** allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.

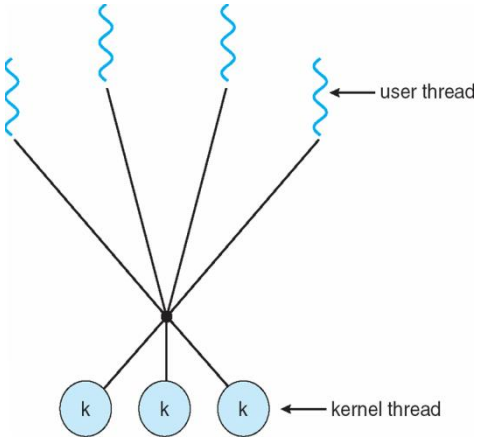
• **The one-to-one model** allows for greater concurrency, but the developer has to be careful not to create too many threads within an application

• **The many-to-many model** suffers from neither of these shortcomings

1) Developers can create as many user threads as necessary

2) The kernel threads can run in parallel on a multiprocessor.

3) When a thread performs a blocking system-call, kernel can schedule another thread for execution.



Two Level Model

• A variation on the many-to-many model is the two level-model as shown in Figure.

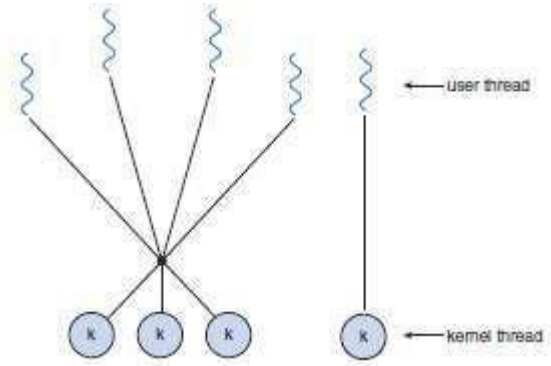
• Similar to **many-to-many model** and also that it allows a user thread to be bound to kernel thread.

• For example:

→ HP-UX

→ Tru64 UNIX

→ IRIX



2.7 Thread Libraries

• It provides the programmer with an API for the creation and management of threads.

• Two ways of implementation:

1) First Approach

-Provides a library entirely in user space with no kernel support.

- All code and data structures for the library exist in the user space.

-This means that invoking a function in the library results in a local function call in user space and not a system call.

2) Second Approach

- Implements a kernel-level library supported directly by the OS.
- Code and data structures for the library exist in kernel space.
- Invoking a function in the API for the library typically results in a system call to the kernel.

• Three main thread libraries:

1) POSIX Pthreads

2) Win32 and

3) Java.

- As an illustrative example, a multithreaded program is designed that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=0}^{N} i$$

- For example, if N were 5, this function would represent the summation from 0 to 5, which is 15.

2.7.1 Pthreads

• This is a POSIX standard API for thread creation and synchronization.
• This is a specification for thread-behavior, not an implementation.
• OS designers may implement the specification in any way they wish.
• Commonly used in: UNIX and Solaris.
• The C program shown in Figure 4.6 demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a nonnegative integer in a separate thread

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }

  /* get the default attributes */
  pthread_attr_init(&attr);
  /* create the thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* wait for the thread to exit */
  pthread_join(tid,NULL);

  printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for (i = 1; i <= upper; i++)
    sum += i;

  pthread_exit(0);
}
```

- In a Pthreads program, separate threads begin execution in a specified function.
- In Figure , this is the runner () function.
- When this program begins, a single thread of control begins in main( ).
- After some initialization, main() creates a second thread that begins control in the runner () function.
- Both threads share the global data sum.
- All Pthreads programs must include the pthread.h header file.
- The statement **pthread_t tid** declares the identifier for the thread we will create.
- Each thread has a set of attributes, including stack size and scheduling information.
- The **pthread.attr_t attr** declaration represents the attributes for the thread.
- We set the attributes in the function Call **pthread_attr_init(&attr)**.
- the default attributes provided
- A separate thread is created with the pthread_create() function call passing the
  - ➢ thread identifier
  - ➢ the attributes for the thread,
  - ➢ the name of the function where the new thread will begin execution (in this case, the runner () function).
  - ➢ the integer parameter that was provided on the command line, argv [1].
- At this point, the program has two threads:
  -the initial (or parent) thread in main( ) and
  -the summation (or child) thread performing the summation operation in the runner () function.
- After creating the summation thread, the parent thread will wait for it to complete by calling the **pthread_join( )** function.
- The summation thread will complete when it calls the function **pthread.exit ( ) .**
- Once the summation thread has returned, the parent thread will output the value of the shared data sum

2.7.2 Win32 Threads

- The Win32 thread API is illustrated in the C program shown in Figure below.
- we must include the **windows.h** header file when using the Win32 API.
- data shared by the separate threads—in this case, Sum—are declared globally
- the **DWORD** data type is an unsigned 32-bit integer.
- the **Summation()** function- is to be performed in a separate thread.
- This function is passed a pointer to a void, which Win32 defines as **LPVOID**.
- The thread performing this function sets the global data Sum to the value of the summation from 0 to the parameter passed to **Summation()**.
- Threads are created in the Win32 API using the **CreateThread()** function and a set of attributes for the thread is passed to this function.
- These attributes include
  - security information,
  - the size of the stack, and
  - a flag that can be set to indicate if the thread is to start in a suspended state.
- In this program, we use the default values for these attributes (which do not initially set the thread to a suspended state and instead make it eligible to be run by the CPU scheduler).
- Once the summation thread is created, the parent must wait for it to complete before outputting the value of Sum, as the value is set by the summation thread.

```
#inciude <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;
   /* perform some basic error checking */
   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }

   // create the thread
   ThreadHandle = CreateThread(
      NULL, // default security attributes
      0, // default stack size
      Summation, // thread function
      &Param, // parameter to thread function
      0, // default creation flags
      &ThreadId); // returns the thread identifier

   if (ThreadHandle != NULL) {
      // now wait for the thread to finish
      WaitForSingleObject(ThreadHandle,INFINITE);

      // close the thread handle
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

- In the Win32 API parent wait using the **WaitForSingleObj ect ()** function, which causes the creating thread to block until the summation thread has exited.

2.7.3 Java Threads

•Threads are the basic model of program-execution in

→ Java program and

→ Java language.

• The API provides a rich set of features for the creation and management of threads.

• All Java programs comprise at least a single thread of control (ex. main()).

• Two techniques for creating threads:

1) Create a new class that is derived from the Thread class and override its run() method.

2) Define a class that implements the Runnable interface. The Runnable interface is defined as follows:

```
public interface Runnable
{
   public abstract void run();
}
```

• When a class implements Runnable, it must define a run() method.

• The code implementing the run() method is what runs as a separate thread.

• Figure shows the Java version of a multithreaded program that determines the summation of a non-negative integer.

• The Summation class implements the Runnable interface.

• Thread creation is performed by creating an object instance of the Thread class and passing the constructor a Runnable object.

• Creating a Thread object does not specifically create the new thread; rather, it is the start () method that actually creates the new thread.

• Calling the start () method for the new object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the run () method, making the thread eligible to be run by the JVM.

• When the summation program runs, two threads are created by the JVM.

1) the parent thread, which starts execution in the main() method.

2) thread is created when the start () method on the Thread object is invoked. This child thread begins execution in the run () method of the Summation class.

• After outputting the value of the summation, this thread terminates when it exits from its run () method.

• if two or more threads are to share data in a Java program, the sharing occurs by passing reference to the shared object to the appropriate threads.

• In this Java program the main thread and the summation thread share the the object instance of the Sum class. This shared object is referenced through the appropriate getSum( ) and setSum( ) methods.

• the parent threads wait for child to finish before proceeding using the join() method

```
class Sum
{
  private int sum;

  public int getSum() {
    return sum;
  }

  public void setSum(int sum) {
    this.sum = sum;
  }
}

class Summation implements Runnable
{
  private int upper;
  private Sum sumValue;

  public Summation(int upper, Sum sumValue) {
    this.upper = upper;
    this.sumValue = sumValue;
  }

  public void run() {
    int sum = 0;
    for (int i = 0; i <= upper; i++)
      sum += i;
    sumValue.setSum(sum);
  }
}


public class Driver
{
  public static void main(String[] args) {
    if (args.length > 0) {
      if (Integer.parseInt(args[0]) < 0)
        System.err.println(args[0] + " must be >= 0.");
      else {
        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);
        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
          thrd.join();
          System.out.println
                  ("The sum of "+upper+" is "+sumObject.getSum());
        } catch (InterruptedException ie) { }
      }
    }
    else
      System.err.println("Usage: Summation <integer value>"); }
}
```

2.8 Threading Issues

2.8.1 fork() and exec() System-calls

• fork() is used to create a separate, duplicate process.

• If one thread in a program calls fork(), then

1) Some systems duplicates all threads and

2) Other systems duplicate only the thread that invoked the fork().

• If a thread invokes the exec(), the program specified in the parameter to exec() will replace the entire process including all threads.

•Which of the two versions of fork( ) to use depends on the application.

•If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec () will replace the process ,hence duplicating only the calling thread is appropriate.

•If, however, the separate process does not call exec( ) after forking, the separate process should duplicate all threads.

2.8.2 Thread Cancellation

• Thread Cancellation is the task of terminating a thread before it has completed.

• A thread that is to be canceled is often referred to as the target thread.

• Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation. One thread immediately terminates the target thread.

2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

• The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.

• This becomes especially troublesome with asynchronous cancellation.

• Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

• With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not.

• This allows a thread to check whether it should be canceled at a point when it can be canceled safely.

• Pthreads refers to such points as **cancellation points.**

2.8.3 Signal Handling

• In UNIX, a signal is used to notify a process that a particular event has occurred.

• All signals follow this pattern:

1) A signal is generated by the occurrence of a certain event.

2) A generated signal is delivered to a process.

3) Once delivered, the signal must be handled.

• A signal handler is used to process signals.

• A signal may be received either synchronously or asynchronously, depending on the source.

1) Synchronous signals

- Delivered to the same process that performed the operation causing the signal.
- E.g. illegal memory access and division by 0.

2) Asynchronous signals

- Generated by an event external to a running process.
- E.g. user terminating a process with specific keystrokes <ctrl><c>.

• Every signal can be handled by one of two possible handlers:

1) A Default Signal Handler

- Run by the kernel when handling the signal.

2) A User-defined Signal Handler

- Overrides the default signal handler.

• In single-threaded programs, delivering signals is simple.

• In multithreaded programs, delivering signals is more complex. Then, the following options exist:

1) Deliver the signal to the thread to which the signal applies.

2) Deliver the signal to every thread in the process.

3) Deliver the signal to certain threads in the process.

4) Assign a specific thread to receive all signals for the process.

• The method for delivering a signal depends on the type of signal generated.

For example,

-synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.

-Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.

• Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.

• Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it.

• However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.

• The standard UNIX function for delivering a signal is kill(aid_t aid, int signal);

-here, we specify the process (aid) to which a particular signal is to be delivered.

• POSIX Pthreads also provides the pthread_kill(pthread_t tid , int signal) function, which allows a signal to be delivered to a specified thread (tid.)

• Windows does not explicitly provide support for signals, they can be emulated using asynchronous procedure calls (APCs).

• The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event.

2.8.4 Thread Pools

•whenever the server receives a request, it creates a separate thread to service the request.

•Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems.

•The first concerns the amount of time required to create the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work.

•The second issue is more troublesome: If we allow all concurrent requests to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust system resources, such as CPU time or memory.

•solution using a thread pool:

The general idea behind a thread pool is to create a number of threads at process startup and place them into a *pool*, where they sit and wait for work.

• Procedure:

1) When a server receives a request, it awakens a thread from the pool.

2) If any thread is available, the request is passed to it for service.

3) Once the service is completed, the thread returns to the pool.

• Advantages:

1) Servicing a request with an existing thread is usually faster than waiting to create a thread.

2) The pool limits the no. of threads that exist at any one point.

• No. of threads in the pool can be based on factors such as

→ no. of CPUs

→ amount of memory and

→ expected no. of concurrent client-requests.

• The Win32 API provides several functions related to thread pools.

• The java.util.concurrent package in Java 1.5 provides a thread pool utility

2.8.5 Thread-Specific Data

• Threads belonging to a process share the data of the process.

• this sharing of data provides one of the benefits of multithreaded programming.

• However, in some circumstances, each thread might need its own copy of certain data.

• such data are called thread-specific data.

For example,

- in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-specific data.

•Most thread libraries—including Win32, Java and Pthreads—provide some form of support for thread-specific data.

2.8.6 Scheduler Activations

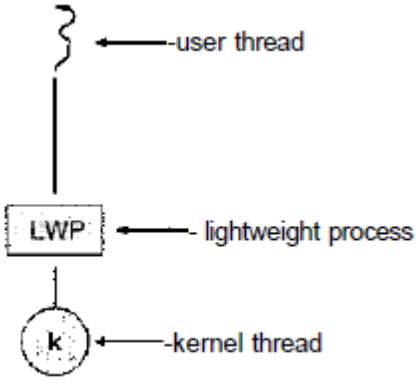
•A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models.

•Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

•Many systems implementing either the many-to-many or two-level model place an intermediate data structure between the user and kernel threads.

•This data structure—typically known as a lightweight process, or LWP—is shown in Figure

•To the user-thread library, the LWP appears to be a *virtual processor* on which the application can schedule a user thread to run.



•Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.

•If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well. •Up the,chain, the user-level thread attached to the LWP also blocks.

•An application may require any number of LWPs to run efficiently.

• a CPU-bound application running on a single processor, one LWP is sufficient.

• An application that is I/Ointensive may require multiple LWPs to execute, however.

•One scheme for communication between the user-thread library and the kernel is known as **scheduler activation.**

•It works as follows:

-The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor.

-the kernel must inform an application about certain events. This procedure is known as an upcall.

-Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.

PROCESS SCHEDULING

2.9 Basic Concepts

• In a single-processor system,

→ only one process may run at a time.

→ other processes must wait until the CPU is rescheduled.

• Objective of multiprogramming:

→ to have some process running at all times, in order to maximize CPU utilization.

2.9.1 CPU-I/0 Burst Cycle

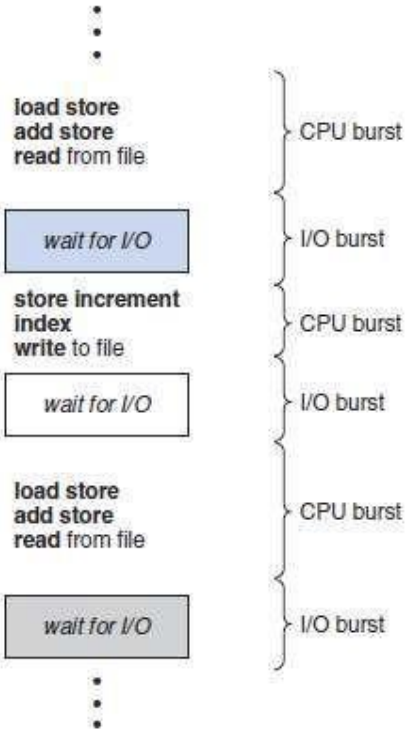
• Process execution consists of a cycle of

→ CPU execution and

→ I/O wait as shown in figure

• Process execution begins with a CPU burst, followed by an I/O burst, then another CPU burst, etc

• Finally, a CPU burst ends with a request to terminate execution. … as shown in figure

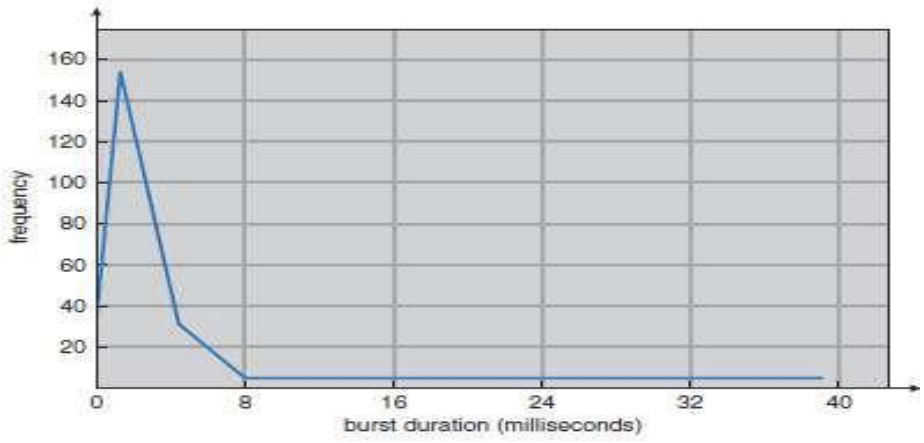


•The durations of CPU bursts vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure below

•The curve is generally characterized as exponential or hyperexponential,

- An I/O-bound program typically has many short CPU bursts.
- A CPU-bound program might have a few long CPU bursts.

• This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.



2.9.2 CPU Scheduler

• This scheduler
→ selects a waiting-process from the ready-queue and
→ allocates CPU to the waiting-process.
• The ready-queue could be a FIFO, priority queue, tree and list.
• The records in the queues are generally process control blocks (PCBs) of the processes.

2.9.3 Preemptive Scheduling

• Four situations under which CPU scheduling decisions take place:
1) When a process switches from the running state to the waiting state.
For ex; I/O request.
2) When a process switches from the running state to the ready state.
For ex: when an interrupt occurs.
3) When a process switches from the waiting state to the ready state.
For ex: completion of I/O.
4) When a process terminates.

•Scheduling under 1 and 4 is non-preemptive.
•Scheduling under 2 and 3 is preemptive.

Non Preemptive Scheduling

• Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either
→ by terminating or
→ by switching to the waiting state.

Preemptive Scheduling

• This is driven by the idea of prioritized computation.
• Processes that are runnable may be temporarily suspended
• Disadvantages:
1) Incurs a cost associated with access to shared-data.
2) Affects the design of the OS kernel.

2.9.4 Dispatcher

• It gives control of the CPU to the process selected by the short-term scheduler.
• The function involves:
1) Switching context
2) Switching to user mode &
3) Jumping to the proper location in the user program to restart that program.
• It should be as fast as possible, since it is invoked during every process switch.
• **Dispatch latency** means the time taken by the dispatcher to
→ stop one process and
→ start another running.

2.10 Scheduling Criteria

• Different CPU-scheduling algorithms
→ have different properties and
→ may favor one class of processes over another.

• Criteria to compare CPU-scheduling algorithms:

1) CPU Utilization

- We must keep the CPU as busy as possible.
-Conceptually, CPU utilization can range from 0 to 100 percent.
- In a real system, it ranges from 40% to 90%.

2) Throughput

-Number of processes completed per time unit.
- For long processes, throughput may be 1 process per hour;
-For short transactions, throughput might be 10 processes per second.

3) Turnaround Time

-The interval from the time of submission of a process to the time of completion.
- Turnaround time is the sum of the periods spent
→ waiting to get into memory
→ waiting in the ready-queue
→ executing on the CPU and
→ doing I/O.

4) Waiting Time

-The amount of time that a process spends waiting in the ready-queue.

5) Response Time

-The time from the submission of a request until the first response is produced.
-The time is generally limited by the speed of the output device.

• it is desirable
→ to maximize CPU utilization and throughput and
→ to minimize turnaround time, waiting time, and response time.

•some circumstances, it is desirable
→ to optimize the minimum or maximum values rather than the average.

2.11 Scheduling Algorithms

• CPU scheduling deals with the problem of deciding which of the processes in the ready-queue is to be allocated the CPU.

• some of the scheduling algorithms:

1) FCFS scheduling (First Come First Served)
2) Round Robin scheduling
3) SJF scheduling (Shortest Job First)
4) SRTF scheduling
5) Priority scheduling
6) Multilevel Queue scheduling and
7) Multilevel Feedback Queue scheduling

Some useful formulas

AT- arrival time
WT –waiting time
CT- completion time
BT- burst time
TAT – turnaround time
RT – response time
FRT – first response time

- **WT = CT – AT – BT**
- **TAT = CT – AT**
- **RT = FRT – AT**
- **For n process:**
  - **AVG. TAT = sum TAT of n process/ n**
  - **AVG. WT = sum WT of n process/n**

2.11.1 FCFS Scheduling

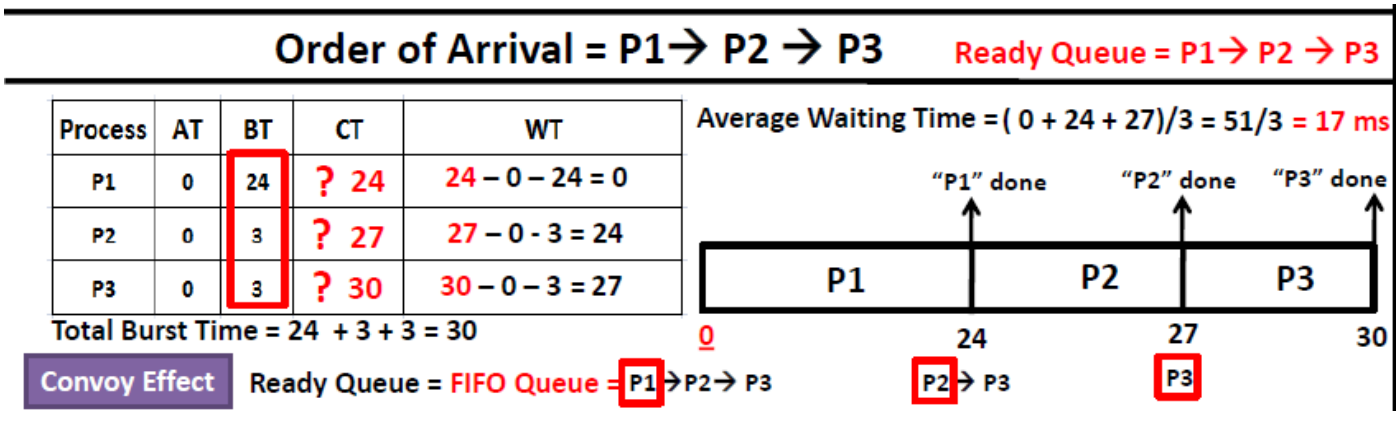
• The process that requests the CPU first is allocated the CPU first.
• The implementation is easily done using a FIFO queue.
• Procedure:
1) When a process enters the ready-queue, its PCB is linked onto the tail of the queue.
2) When the CPU is free, the CPU is allocated to the process at the queue‘s head.
3) The running process is then removed from the queue.
• Advantage:
1) Code is simple to write & understand.
• Disadvantages:
1) **Convoy effect**: All other processes wait for one big process to get off the CPU.
2) Non-preemptive (a process keeps the CPU until it releases it).
3) Not good for time-sharing systems.
4) The average waiting time is generally long

• Example:

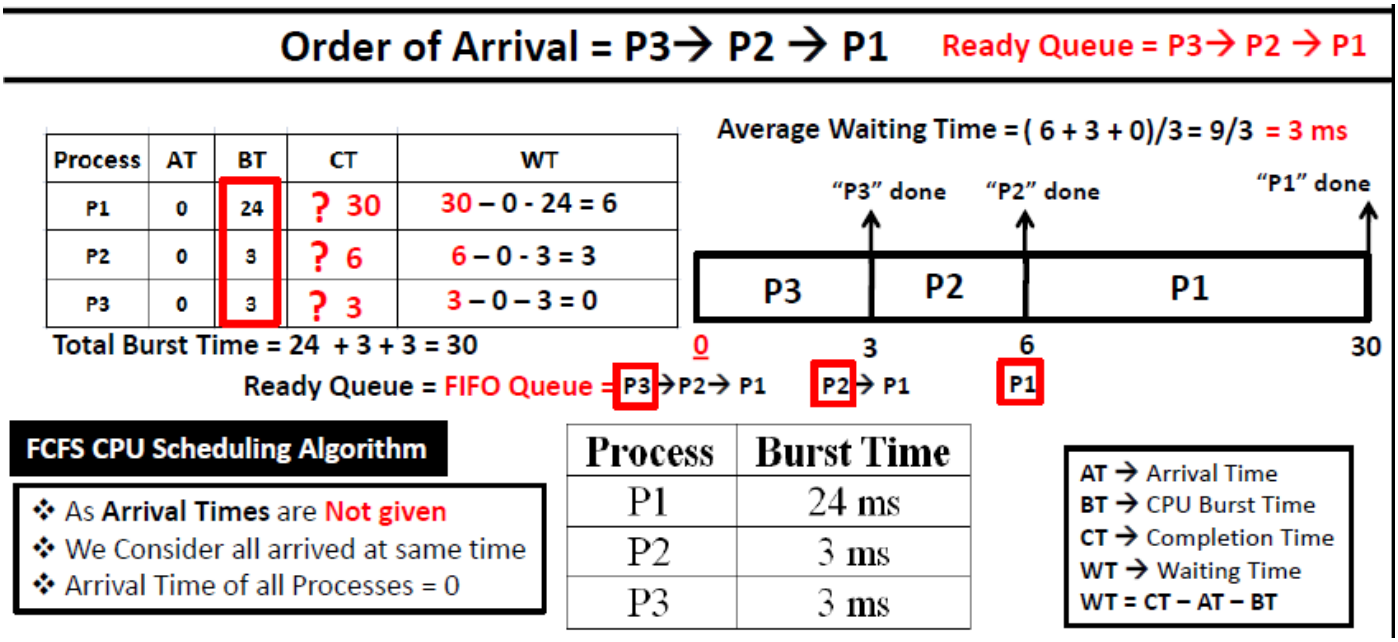
Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

i) Suppose that the processes arrive in the order P1, P2, P3.


Order of Arrival = P1→ P2 → P3
Ready Queue = P1→ P2 → P3
Process AT BT CT WT
P1 0 24 ? 24 24 – 0 – 24 = 0
P2 0 3 ? 27 27 – 0 - 3 = 24
P3 0 3 ? 30 30 – 0 – 3 = 27
Total Burst Time = 24 + 3 + 3 = 30
Average Waiting Time = (0 + 24 + 27)/3 = 51/3 = 17 ms
"P1" done
"P2" done
"P3" done
P1
P2
P3
0
24
27
30
Convoy Effect
Ready Queue = FIFO Queue = P1→P2→ P3
P2→ P3
P3

ii) Suppose that the processes arrive in the order P3, P2, P1.


Order of Arrival = P3→ P2 → P1
Ready Queue = P3→ P2 → P1
Process AT BT CT WT
P1 0 24 ? 30 30 – 0 - 24 = 6
P2 0 3 ? 6 6 – 0 - 3 = 3
P3 0 3 ? 3 3 – 0 – 3 = 0
Total Burst Time = 24 + 3 + 3 = 30
Average Waiting Time = (6 + 3 + 0)/3 = 9/3 = 3 ms
"P3" done
"P2" done
"P1" done
P3
P2
P1
0
3
6
30
Ready Queue = FIFO Queue = P3→P2→ P1
P2→ P1
P1
FCFS CPU Scheduling Algorithm
As Arrival Times are Not given
We Consider all arrived at same time
Arrival Time of all Processes = 0
Process Burst Time
P1 24 ms
P2 3 ms
P3 3 ms
AT → Arrival Time
BT → CPU Burst Time
CT → Completion Time
WT → Waiting Time
WT = CT – AT – BT

[what is Convoy effect?

- Assume we have one CPU-bound process and many I/O-bound processes.
- The CPU-bound process will get and hold the CPU.
- During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.
- While the processes wait in the ready queue, the I/O devices are idle.
- Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device.
- All the I/O-bound processes, which have short CPU bursts, execute quickly and move back to the I/O queues. At this point, the CPU sits idle.
- The CPU-bound process will then move back to the ready queue and be allocated the CPU.
- Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. This is a convoy effect as all the other processes wait for the one big process to get off the CPU.]

2.11.2 SJF Scheduling

• The CPU is assigned to the process that has the smallest next CPU burst.
• If two processes have the same length CPU burst, FCFS scheduling is used to break the tie.
• For long-term scheduling in a batch system, we can use the process time limit specified by the user, as the _length'
• SJF can't be implemented at the level of short-term scheduling, because there is no way to know the length of the next CPU burst
• Advantage:

1) The SJF is optimal, i.e. it gives the minimum average waiting time for a given set of processes.

• Disadvantage:

1) Determining the length of the next CPU burst.

•The next CPU burst is generally predicted as an exponential average of the measured lengths of previous CPU bursts.

• Let *tn* be the length of the n^{th} CPU burst, and let T_{n+1} be our predicted value for the next CPU burst. Then, for α, 0 < α < 1, define

$$T\ n + 1 = \alpha\ tn + (1 - \alpha)Tn.$$

•This formula defines an exponential average.

•The value of tn contains our most recent information; in stores the past history.

•The parameter α controls the relative weight of recent and past history in our prediction.

- •If α = 0, then Tn+1 = Tn and recent history has no effect
- • if α = 1, then Tn+1 = tn, and only the most recent CPU burst matters (history is assumed to be old and irrelevant).
- •More commonly, α = 1/2, so recent history and past history are equally weighted.

• SJF algorithm may be either **- non-preemptive or**

- preemptive.

- When a new process arrives at the ready queue while a previous process is still executing.
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process.

-In Non preemptive SJF

- The current process is allowed to finish its CPU burst.

-In Preemptive SJF

If the new process has a shorter next CPU burst than what is left of the executing process, A preemptive SJF algorithm will preempt the currently executing process.

-It is also known **as SRTF scheduling** (Shortest-Remaining-Time-First).

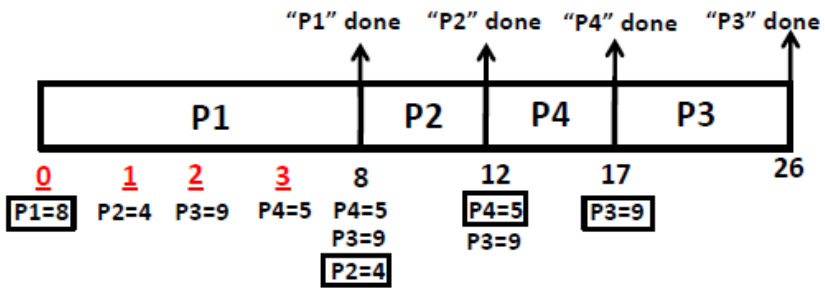
As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Non Pre-emptive SJF

Scheduling → To be done at only Process Completion

Process	AT	BT	CT	WT
P1	0	8	? 8	8 – 8 - 0 = 0
P2	1	4	? 12	12 – 4 – 1=7
P3	2	9	? 26	26– 9 – 2=15
P4	3	5	? 17	17– 5 - 3= 9

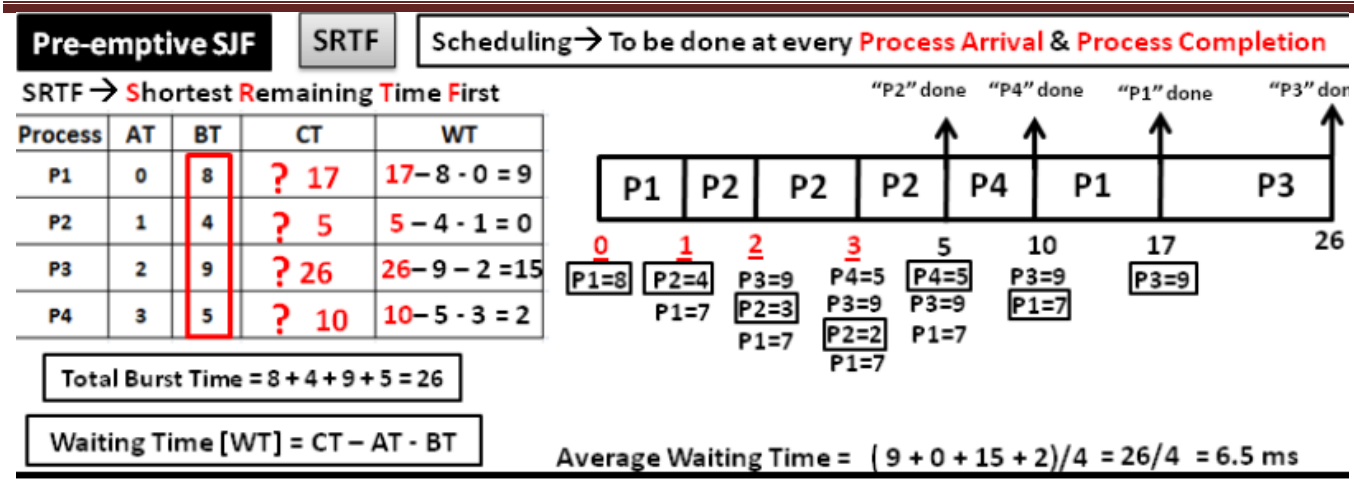


AT → Arrival Time
BT → CPU Burst Time
CT → Completion Time
WT → Waiting Time
TAT → Turn Around Time

WT = CT – AT – BT
TAT = CT - AT

Pre-emptive SJF is always better than Non Pre-emptive SJF

Average Waiting Time = (0 + 7 + 15 + 9)/4 = 31/4 = 7.75 ms


Pre-emptive SJF
SRTF
Scheduling → To be done at every Process Arrival & Process Completion
SRTF → Shortest Remaining Time First
Process AT BT CT WT
P1 0 8 ? 17 17 – 8 - 0 = 9
P2 1 4 ? 5 5 – 4 - 1 = 0
P3 2 9 ? 26 26 – 9 – 2 = 15
P4 3 5 ? 10 10 – 5 - 3 = 2
Total Burst Time = 8 + 4 + 9 + 5 = 26
Waiting Time [WT] = CT – AT - BT
"P2" done
"P4" done
"P1" done
"P3" done
P1 P2 P2 P2 P4 P1 P3
0 1 2 3 5 10 17 26
P1=8
P2=4
P1=7
P3=9
P2=3
P1=7
P4=5
P3=9
P2=2
P1=7
P4=5
P3=9
P1=7
P3=9
P1=7
P3=9
Average Waiting Time = (9 + 0 + 15 + 2)/4 = 26/4 = 6.5 ms

2.11.3 Priority Scheduling

• A priority is associated with each process.
• The CPU is allocated to the process with the highest priority.
• Equal-priority processes are scheduled in FCFS order.
• Priorities can be defined either internally or externally.

1) Internally-defined priorities.
- Use some measurable quantity to compute the priority of a process.
- For example: time limits, memory requirements, no. of open files.
2) Externally-defined priorities.
- Set by criteria that are external to the OS
- For example: importance of the process, political factors

• Priority scheduling can be either preemptive or nonpreemptive.

1) Preemptive
- The CPU is preempted if the priority of the newly arrived process is higher than the priority of the currently running process.
2) Non Preemptive
- The new process is put at the head of the ready-queue

• Advantage:

1) Higher priority processes can be executed first.

• Disadvantage:

1**) Indefinite blocking or starvation** , where low-priority processes are left waiting indefinitely for CPU.

Solution: Aging is a technique of increasing priority of processes that wait in system for a long time.

Illustration of Aging

1) Priorities in system = {0, 1, 2, ... , 10} with **High Priority = 10** and **Low Priority = 0**
2) **Aging** → Increase priority of process by 1 for every 1 hour waiting in Ready Queue.
3) Process with Priority = 0 → Will get Priority = 10 after **waiting in Ready Queue** for 10 hours.

Example, consider the following set of processes, assumed to have arrived at time 0, in the order P1, P2, • • -, *P5*, with the length of the CPU burst given in milliseconds:

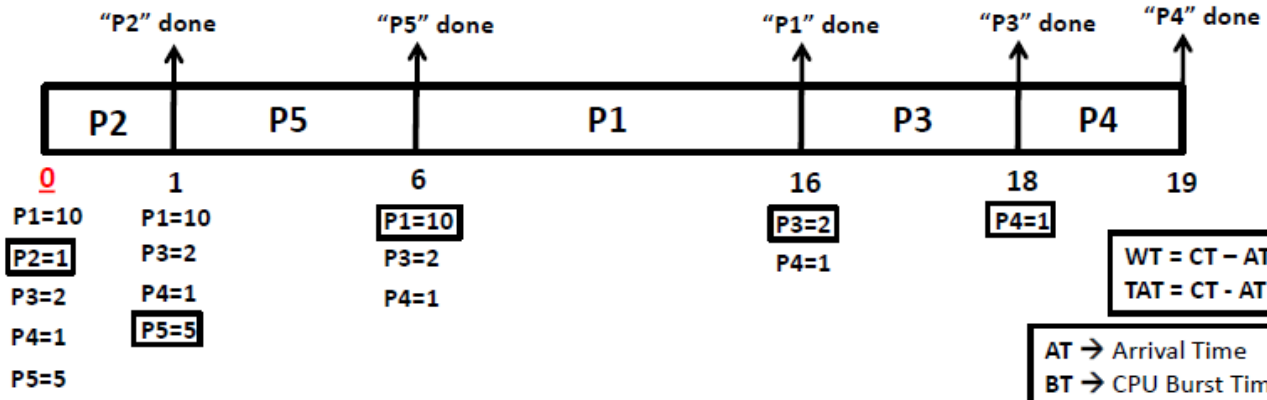
Assume that low numbers represent high priority

Process	Burst Time	Priority
P1	10 ms	3
P2	1 ms	1
P3	2 ms	4
P4	1 ms	5
P5	5 ms	2

Process	AT	BT	Priority	CT	WT
P1	0	10		? 16	16 – 10 - 0 = 6
P2	0	1		? 1	1 – 1 - 0 = 0
P3	0	2		? 18	18 – 2 – 0 = 16
P4	0	1		? 19	19 – 1 - 0 = 18
P5	0	5		? 6	6 – 5 - 0 = 1

Priority Queue = P2 → P5 → P1 → P3 → P4

Total Burst Time = 1 + 5 + 10 + 2 + 1 = 19



WT = CT – AT – BT
TAT = CT - AT

AT → Arrival Time
BT → CPU Burst Time
CT → Completion Time
WT → Waiting Time
TAT → Turn Around Time

Average Waiting Time = (6 + 0 + 16 + 18 + 1)/5 = 41/5 = 8.2 ms

2.11.4 Round Robin Scheduling

• Designed especially for timesharing systems.
• It is similar to FCFS scheduling, but with preemption.
• A small unit of time is called a **time quantum (or time slice).**
• Time quantum is ranges from 10 to 100 ms.
• The ready-queue is treated as a **circular queue.**
• **Among all CPU Scheduling Algorithms, Round Robin will give Minimum Response Time.**
• The CPU scheduler
→ goes around the ready-queue and
→ allocates the CPU to each process for a time interval of up to 1 time quantum.
• To implement:
The ready-queue is kept as a FIFO queue of processes
• CPU scheduler
1) Picks the first process from the ready-queue.
2) Sets a timer to interrupt after 1 time quantum and
3) Dispatches the process.
• One of two things will then happen.
1) The process may have a CPU burst of less than 1 time quantum.
In this case, the process itself will release the CPU voluntarily.

2) If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the OS.

-that process is *preempted* and is put back in the ready queue at tail.

The RR scheduling algorithm is thus preemptive.

•disadvantage:

1) Higher average turnaround than SJF.

•advantage:

1) Better response time than SJF.

Example 1:

Compute Average Waiting Time & Average Response Time using RR with quantum time q=4 for the following process:

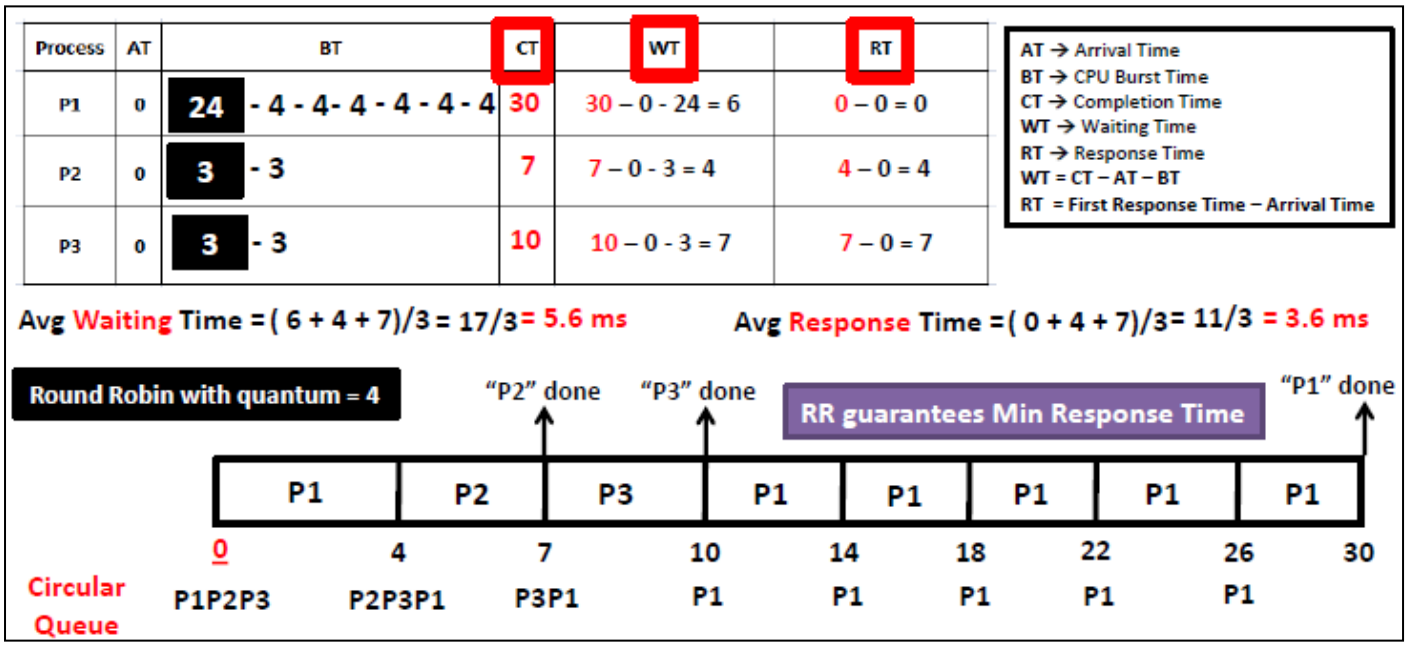
Process	Burst Time
P1	24 ms
P2	3 ms
P3	3 ms

Process	AT	BT	CT	WT	RT
P1	0	24 - 4 - 4 - 4 - 4 - 4 - 4	30	30 – 0 - 24 = 6	0 – 0 = 0
P2	0	3 - 3	7	7 – 0 - 3 = 4	4 – 0 = 4
P3	0	3 - 3	10	10 – 0 - 3 = 7	7 – 0 = 7

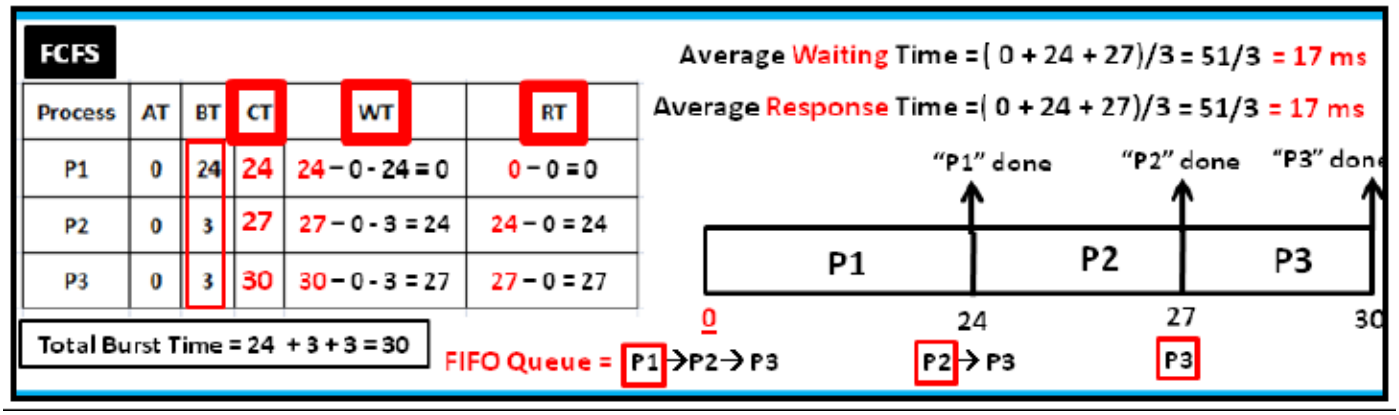
AT → Arrival Time
BT → CPU Burst Time
CT → Completion Time
WT → Waiting Time
RT → Response Time
WT = CT – AT – BT
RT = First Response Time – Arrival Time

Avg Waiting Time = (6 + 4 + 7)/3 = 17/3 = 5.6 ms

Avg Response Time = (0 + 4 + 7)/3 = 11/3 = 3.6 ms



ii) **Above example using FCFS**

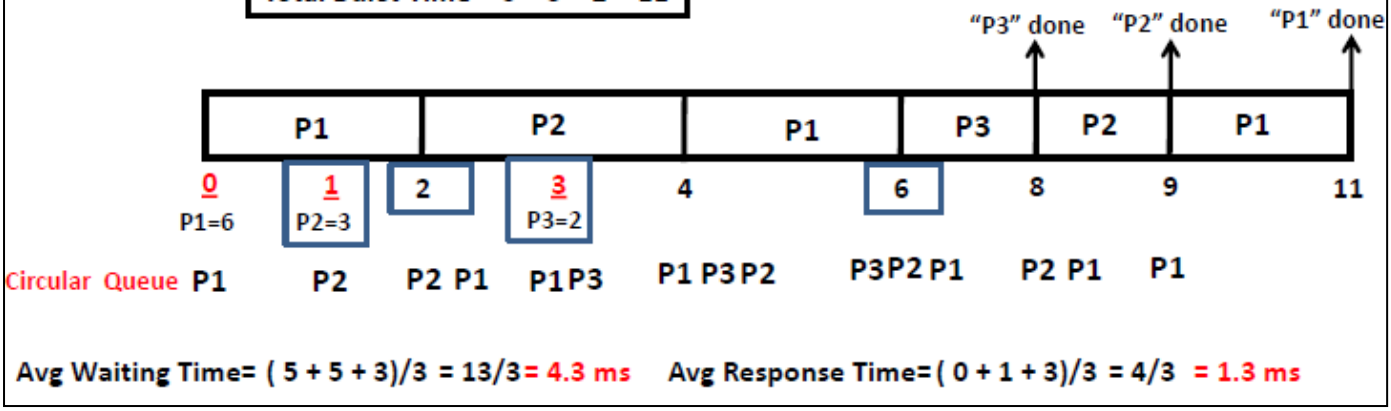

FCFS
Average Waiting Time = (0 + 24 + 27)/3 = 51/3 = 17 ms
Average Response Time = (0 + 24 + 27)/3 = 51/3 = 17 ms
Process AT BT CT WT RT
P1 0 24 24 24 – 0 - 24 = 0 0 – 0 = 0
P2 0 3 27 27 – 0 - 3 = 24 24 – 0 = 24
P3 0 3 30 30 – 0 - 3 = 27 27 – 0 = 27
Total Burst Time = 24 + 3 + 3 = 30
FIFO Queue = P1 → P2 → P3
"P1" done
"P2" done
"P3" done
P1
P2
P3
0
24
27
30
P2 → P3
P3

Example2:

Round Robin with quantum =2 msec With different Arrival Times of Processes

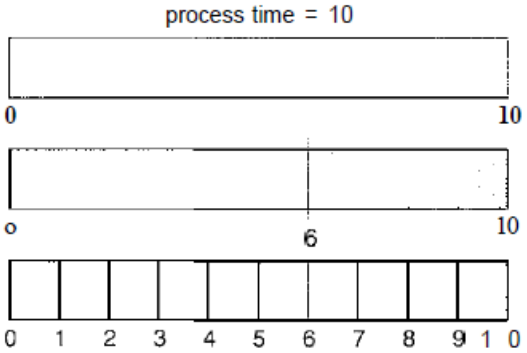
Process	AT	BT	CT	WT	RT
P1	0	6 - 2 - 2 - 2	11	11 – 0 - 6 = 5	0 – 0 = 0
P2	1	3 - 2 - 1	9	9 – 1 - 3 = 5	2 – 1 = 1
P3	3	2 - 2	8	8 – 3 - 2 = 3	6 – 3 = 3

Total Burst Time = 6 + 3 + 2 = 11



Avg Waiting Time= (5 + 5 + 3)/3 = 13/3 = 4.3 ms Avg Response Time= (0 + 1 + 3)/3 = 4/3 = 1.3 ms

- **No of Processes in Ready Queue = n**
- Size of Quantum = **q**
- All processes will get a quantum each in **n × q** time units. [First Cycle completes]
- I have **just completed** my quantum;
- I will get another quantum after **(n-1) × q** time units.
- Performance of RR depends
  - on size of quantum
    - Quantum is large – RR will become FCFS
    - Quantum is small – RR will become Processor Sharing
  - context switching time.



quantum	context switches
12	0
6	1
1	9

2.11.5 Multilevel Queue Scheduling

• Useful for situations in which processes are easily classified into different groups.

• For example, a common division is made between

→ foreground (or interactive) processes and

→ background (or batch) processes.

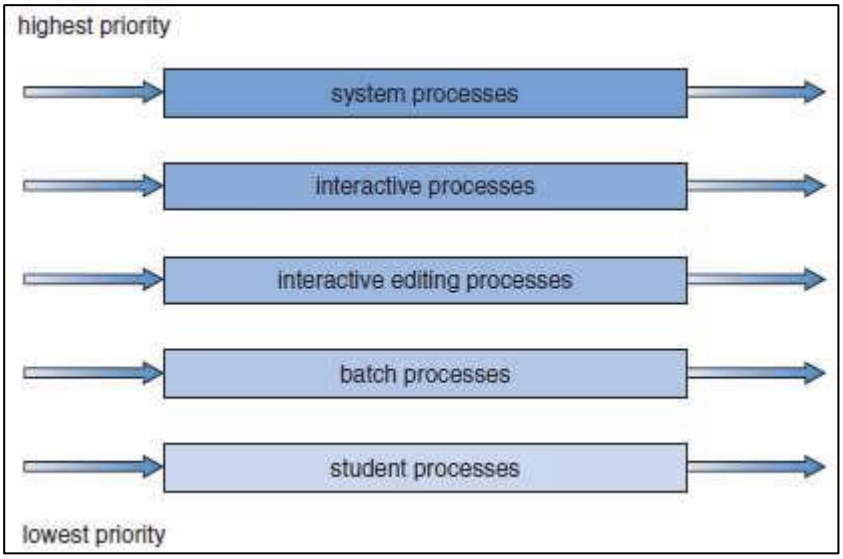
• The ready-queue is partitioned into several separate queues (as shown in Figure below).

• The processes are permanently assigned to one queue based on some property like

→ memory size

→ process priority or

→ process type.



• Each queue has its own scheduling algorithm.

For example, -separate queues might be used for foreground and background processes.

-The foreground quetie might be scheduled by an RR algorithm, while the background

-queue is scheduled by an FCFS algorithm.

• Scheduling can be done in 2 ways:-

1) Each queue has absolute priority over lower-priority queues.

-No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty.

-If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

2) **Time slice:** each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR ,20% to background in FCFS

2.11.6 Multilevel Feedback Queue Scheduling

• A process may move between queues .

• The basic idea is :

To Separate processes according to the features of their CPU bursts. For example

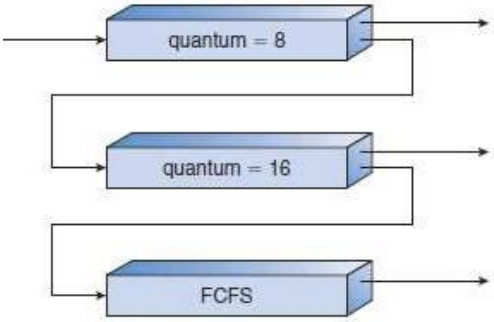
1) If a process uses too much CPU time, it will be moved to a lower-priority queue.

-This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

2) If a process waits too long in a lower-priority queue, it may be moved to a higher priority queue .

- This form of aging prevents starvation.

•For example, consider a multilevel feedback-queue scheduler with three queues, numbered from 0 to 2 as shown in figure.



•The scheduler first executes all processes in queue 0.

•Only when queue 0 is empty will it execute processes in queue 1.

•Similarly, processes in queue 2 will only be executed if queues 0 and 1 are empty.

• A process that arrives for queue 1 will preempt a process in queue 2.

•A process in queue 1 will in turn be preempted by a process arriving for queue 0.

•A process entering the ready queue is put in queue 0.

•A process in queue 0 is given a time quantum of 8 milliseconds.

•If it does not finish within this time, it is moved to the tail of queue 1.

•If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds.

•If it does not complete, it is preempted and is put into queue 2.

•Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty.

•This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less.

•Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.

•Processes that need more than 8 but less than 24 milliseconds(8+16) are also served quickly, although with lower priority than shorter processes.

•Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.

• In general, a multilevel feedback queue scheduler is defined by the following parameters:

1) The number of queues.

2) The scheduling algorithm for each queue.

3) The method used to determine when to upgrade a process to a higher priority queue.

4) The method used to determine when to demote a process to a lower priority queue.

5) The method used to determine which queue a process will enter when that process needs service.

2.12 Multiple Processor Scheduling

• If multiple CPUs are available, the scheduling problem becomes more complex.

• Two approaches:

1) Asymmetric Multiprocessing

The basic idea is:

i) A master server is a single processor responsible for all scheduling decisions, I/O processing and other system activities.

ii) The other processors execute only user code.

Advantage:

-This is simple because only one processor accesses the system data structures, reducing the need for data sharing.

2) Symmetric Multiprocessing

The basic idea is:

i) Each processor is self-scheduling.

ii) All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.

ii) To do scheduling, the scheduler for each processor

i. Examines the ready-queue and

ii. Selects a process to execute.

We must ensure that two processors do not choose the same process and that processes are not lost from the queue.

2.12.1 Processor Affinity

• The data most recently accessed by the process populates the cache for the processor; and as a result, successive memory accesses by the process are often satisfied in cache memory.

• If the process migrates to another processor:

-The contents of cache memory must be invalidated for the processor being migrated from, and

-the cache for the processor being migrated to must be re-populated.

Because of the high cost of invalidating and re-populating caches, in SMP systems,

1) Migration of processes from one processor to another are avoided and

2) Instead processes are kept running on same processor. This is known as **processor affinity.**

• Two forms:

1) Soft Affinity

- When an OS try to keep a process on one processor because of policy, but cannot guarantee it will happen.

- It is possible for a process to migrate between processors.

2) Hard Affinity

- When an OS have the ability to allow a process to specify that it is not to migrate to other processors. Eg: Solaris OS

2.12.2 Load Balancing

• This attempts to keep the workload evenly distributed across all processors in an SMP system.

• Two approaches:

1) Push Migration

-A specific task periodically checks the load on each processor and if it finds an imbalance, it evenly distributes the load by moving processes to idle or less busy processors.

2) Pull Migration

- An idle processor pulls a waiting task from a busy processor.

•Push and pull migration need not be mutually exclusive and are in fact often implemented in parallel on load-balancing systems.

•load balancing often counteracts the benefits of processor affinity

2.12.3 Symmetric Multithreading

• The basic idea:

1) Create multiple logical processors on the same physical processor.

2) Present a view of several logical processors to the OS.

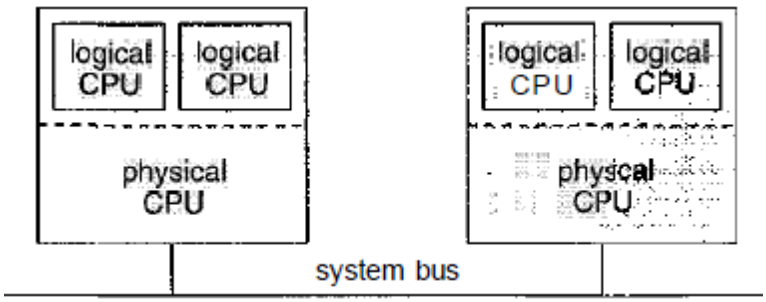
• Such a strategy is known as **symmetric multithreading (SMT) or hyperthreading technology** on Intel processors.

• Each logical processor has its own architecture state, which includes general-purpose and machine state registers.

• Each logical processor is responsible for its own interrupt handling.

• SMT is a feature provided in hardware, not software.

• Figure illustrates a typical SMT architecture with two physical processors, each housing two logical processors.



• From the operating system's perspective, four processors are available for work on this system.

• Performance gains are possible if the operating system is aware that it is running on such a system.

• For example,

-consider a system with two physical processors, both of which are idle.

-The scheduler should first try scheduling separate threads on each physical processor rather than on separate logical processors on the same physical processor.

-Otherwise, both logical processors on one physical processor could be busy while the other physical processor remained idle.

2.13 Thread Scheduling

• On Oss that support threads, it is kernel-level threads but not processes that are being scheduled by the OS.

• User-level threads are managed by a thread library, and the kernel is unaware of them.

• To run on a CPU, user-level threads must be mapped to an associated kernel-level thread.

2.13.1 Contention Scope

1) Process-Contention scope (PCS)

- On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This is known as **Process-Contention scope**

- Competition for the CPU takes place among threads belonging to the same process.

2) System-Contention scope (SCS)

-scheduling a user thread to a LWP does not mean that the thread is running on CPU.

-we require the kernel thread to be scheduled on CPU

-The process of deciding which kernel thread to schedule on the CPU is known as **System-Contention scope**

-Competition for the CPU takes place among all threads in the system.

-Systems using the one-to-one model schedule threads using only SCS.

•PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run.

•User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread.

• It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread;

2.13.2 Pthread Scheduling

• Pthread API allows specifying either PCS or SCS during thread creation.

• Pthreads identifies the following contention scope values:

1) PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling.

2) PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

• Pthread IPC provides following two functions for getting and setting the contention scope policy:

1) pthread_attr_setscope(pthread_attr_t *attr, int scope)

2) pthread_attr_getscope(pthread_attr_t *attr, int *scope)

•The first parameter for both functions contains a pointer to the attribute set for the thread.

•The second parameter for the pthread_attr_setscope ( ) function is passed either the PTHREAD_SCOPE_SYSTEM or PTHREAD_SCOPE_PROCESS value, indicating how the contention scope is to be set.

•In pthread_attr_getscope( ), the second parameter contains a pointer to an int value that is set to the current value of the contention scope.

•If an error occurs, each of these functions returns non-zero values.

•a Pthread program here that first determines the existing contention scope and sets it to PTHREAD_SCOPE_SYSTEM.

•It then creates five separate threads that will run using the SCS scheduling policy.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, scope;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
       printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
       printf("PTHREAD_SCOPE_SYSTEM");
      else
       fprintf(stderr, "Illegal scope value.\n");
    }

   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

MODULE 3
PROCESS SYNCHRONIZATION

3.1 Synchronization

• Co-operating process is one that can affect or be affected by other processes.
• Co-operating processes may either
→ share a logical address-space (i.e. code & data) or →
share data through files or
→ messages through threads.
• Concurrent-access to shared-data may result in data-inconsistency.
• hence synchronization is required among co-operating process.
• To maintain data-consistency:
-The orderly execution of co-operating processes is necessary.
• A solution to producer-consumer problem that fills all buffers.
-By using a variable counter that keeps track of the number of full buffers.
-Initially, counter=0.
- counter is incremented by the producer after it produces a new element in the buffer.
- counter is decremented by the consumer after it consumes a element from the buffer.

- **The code for the producer process is as follows:**

```
while (true)
{
     /* produce an item in nextProduced */
     while (counter == BUFFER_SIZE)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

- **The code for the consumer process as follows**

```
while (true)
{
     while (counter == 0)
        ; /* do nothing */
     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in nextConsumed */
}
```

• Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently**.**

• Example:
  • Counter++ could be implemented as:

$register_1$ = counter
$register_1$ = $register_1$ + 1
counter = $register_1$

  • counter-- may be implemented as:

$register_2$ = counter
$register_2$ = $register_2$ - 1
counter = $register_2$

• The concurrent execution of "counter++" and "counter--" is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved).

• One such interleaving is with counter=5

T_0:	*producer*	execute	$register_1$ = counter	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2$ = counter	$\{register2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	counter = $register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	counter = $register_2$	$\{counter = 4\}$

• The value of counter may be either 4 or 6, where the correct result should be 5. This is an example for race condition.

• Race condition:

A situation where several processes access & manipulate same data concurrently and the outcome of the execution depends on particular order in which the access takes place, is called a **race condition.**

• To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter.

• To make such a guarantee, we require that the processes be synchronized in some way.

3.2 Critical-Section Problem

• Consider a system consisting of *n* processes {P0, P1, ..., Pn }.

• Each process has a segment of code, called a critical section, in which a process may be
→ changing common variables
→ updating a table or
→ writing a file.

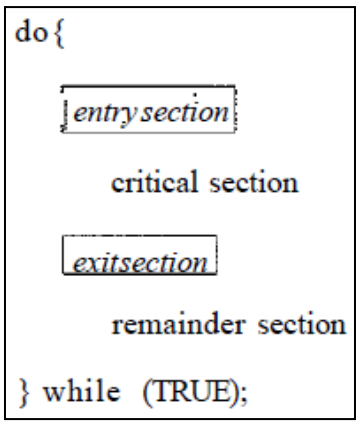
• General structure of a typical process has following (as shown in figure):

1) Entry-section
-Requests permission to enter the critical-section.

2) Critical-section
-Mutually exclusive in time ,
i.e. no other process can execute in its critical-section.

3) Exit-section
-Follows the critical-section.

4) Remainder-section



A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections**.**

2. Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

3. Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

•Race conditions may occur in Kernnel Mode process.

• Two approaches used to handle critical-sections:

1) Preemptive Kernels

-Allows a process to be preempted while it is running in kernel-mode.

2) Non-preemptive Kernels

-Does not allow a process running in kernel-mode to be preempted.

-a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

- We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions.

3.3 Peterson's Solution

• This is a classic software-based solution to the critical-section problem.

• This is limited to 2 processes.

• The 2 processes alternate execution between

→ critical-sections and

→ remainder-sections.

• The 2 processes share 2 variables

int turn;

boolean flag[2];

where

- **turn** = indicates whose turn it is to enter its critical-section. (i.e., if turn==i, then process Pi is allowed to execute in its critical-section).
- **flag** = used to indicate if a process is ready to enter its critical-section. (i.e. if flag[i]=true, then Pi is ready to enter its critical-section).

• The structure of process Pi in Peterson's solution follows:

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

• To enter the critical-section,

→ firstly process Pi sets flag[i] to be true and

→ then sets turn to the value j.

• If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

• The final value of turn determines which of the 2 processes is allowed to enter its critical-section first.

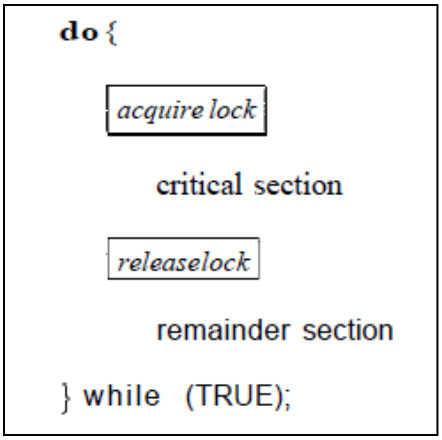
• With this solution :

1) Mutual-exclusion is preserved.

2) The progress requirement is satisfied.

3) The bounded-waiting requirement is met.

3.4 Synchronization Hardware



• Hardware based Solution for Critical-section Problem.
• A **lock** is a simple tool used to solve the critical-section problem.
• Race conditions are prevented by following restriction as shown in figure.
- A process must acquire a lock before entering a criticalsection.
- The process releases the lock when it exits the critical-section.
• Many systems provide hardware support for critical section code.
• Uniprocessors – could disable interrupts
  • Currently running code would execute without preemption
  • Generally too inefficient on multiprocessor systems
  • Operating systems using this not broadly scalable

• Modern machines provide special atomic hardware instructions
  • Either test memory word and set value
  • Or swap contents of two memory words

(Atomic-operation means an operation that completes in its entirety without interruption.)

3.4.1. TestAndSet ( )

The TestAndSet() instruction can be defined as shown in below

```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

This function sets target to true and returns the previous value of target(i.e value that was passed)

• The important characteristic is that this instruction is executed atomically.
• Thus, if two TestAndSet ( ) instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.
• If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.
• The structure of process Pi is shown in below:

```
do {
   while (TestAndSetLock(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

3.4.2. Swap ( )

• The Swap ( ) instruction, in contrast to the TestAndSet ( ) instruction, operates on the contents of two words; it is defined as shown

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

• It is executed atomically.

• If the machine supports the Swap( ) instruction, then mutual exclusion can be provided as follows.

- A global Boolean variable lock is declared and is initialized to false.

-In addition, each process has a local Boolean variable key.

• The structure of process *Pi* is shown here

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock,&key);

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

3.4.3. The TestAndSet ( ) instruction that satisfies all the critical-section requirements.

•The common data structures are

```
boolean waiting[n];
boolean lock;
```

•These data structures are initialized to false.

• process Pi can enter its critical section only if either waiting[i] = = false or key = = false.

• The value of key can become false only if the TestAndSet ( ) is executed.

• The first process to execute the TestAndSet ( ) will find key = = false; all others must wait.

• The variable waiting[i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, thus **maintaining the mutual-exclusion requirement and progress.**

• when a process leaves its critical section, it scans the array waiting in the cyclic ordering
(i + 1, i + 2,..., n-1, 0, ..., i-1).

• It designates the first process in this ordering that is in the entry section (waiting [j] = = true) as the next one to enter the critical section by making waiting [j] =- FALSE

• Thus **the bounded-waiting requirement is met**

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) &&  !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
}while (TRUE);
```

3.5 Semaphores

• The various hardware-based solutions to the critical-section problem are complicated for application programmers to use.

• so we can use a semaphore which is a synchronization-tool.

• It used to control access to shared-variables so that only one process may at any point in time change the value of the shared-variable.

• A semaphore(S) is an integer-variable that is **apart from initialization** accessed only through 2 atomic-operations:

1) wait( ) - termed as P ("to test"). Proberen

2) signal( ) -termed as V ("to increment"). Verhogen

• Definition of wait( ) :

```
wait(S) {
    while S <=0
       ; // no-op
    S--;
}
```

Definition of signal( ):

```
signal(S) {
    S++;
}
```

• When one process modifies the semaphore-value, no other process can simultaneously modify that same semaphore-value.

3.5.1 Semaphore Usage

• Operating systems often distinguish between counting and binary semaphores

- **Counting Semaphore**
  - The value of a semaphore can range over an unrestricted domain
- **Binary Semaphore**
  - The value of a semaphore can range only between 0 and 1.
  - On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual-exclusion.

➢ **Solution for Critical-section Problem using Binary Semaphores**

• Binary semaphores can be used to solve the critical-section problem for multiple processes.

• The n processes share a semaphore mutex, initialized to 1.

• Each process P, is organized as shown in Figure below:

```
do {
   waiting(mutex);
      // critical section
   signal(mutex);
      // remainder section
}while (TRUE);
```

➢ **Use of Counting Semaphores**

• Counting semaphores can be used to control access to a given resource consisting of a finite number o£ instances.

• The semaphore is initialized to the number of resources available.

• Each process that wishes to use a resource performs a wait( ) operation on the semaphore (thereby decrementing the count).

• When a process releases a resource, it performs a signal( ) operation (incrementing the count).

• When the count for the semaphore goes to 0, all resources are being used.

• After that, processes that wish to use a resource will block until the count becomes greater than 0.

➢ **Solving Synchronization Problems**

• Semaphores can also be used to solve synchronization problems.

• For example, consider 2 concurrently running-processes:

-P1 with a statement S1 and

-P2 with a statement S2.

• Suppose we require that S2 be executed only after S1 has completed.

• We can implement this scheme readily

→ by letting P1 and P2 share a common semaphore **synch** initialized to 0, and →

by inserting the following statements in process P1

```
S1;
signal(synch);
```

and the following statements in process P2

```
wait(synch);
S2;
```

• Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

3.5.2 Semaphore Implementation

• Main disadvantage of semaphore: Busy waiting.

• Busy waiting: While a process is in its critical-section, any other process that tries to enter its critical-section must loop continuously in the entry code.

• Busy waiting wastes CPU cycles that some other process might be able to use productively.

• This type of semaphore is also called a spinlock (because the process "spins" while waiting for the lock).

• To overcome busy waiting, we can modify the definition of the wait() and signal() as follows:

1) When a process executes the wait( ) and finds that the semaphore-value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself.

2) A process that is blocked (waiting on a semaphore S) should be restarted when some other process executes a signal(). The process is restarted by a wakeup( ).

• to achieve the above 2 simple operations are used:

1) block( ) suspends the process that invokes it.

2) wakeup(P) resumes the execution of a blocked process P.

These two operations are provided by the operating system as basic system calls.

• Hence a semaphore is defined as below :

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

• The wait( ) semaphore operation can be defined as

```
wait(semaphore *S) {
            S->value--;
            if (S->value < 0) {
                    add this process to S->list;
                    block();
            }
}
```

• The signal( ) semaphore operation can now be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

• The critical aspect of semaphores is that they be executed atomically.

• This problem can be solved in two ways:

- In a uni-processor environment
  - -Inhibit interrupts when the wait and signal operations execute.
  - -Only current process executes, until interrupts are re-enabled & the scheduler regains control.
- In a multiprocessor environment
  - - Inhibiting interrupts doesn't work. So we can use the hardware / software solutions described before.

3.5.3 Deadlocks & Starvation

• The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes

• The event in question is the execution of a signal() operation.

• When such a state is reached, these processes are said to be deadlocked.

• To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

• Suppose that *P0* executes wait (S) and then P1 executes wait (Q).

• When P0 executes wait(Q), it must wait until P1 executes signal(Q).

• Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S).

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

• Since these signal( ) operations cannot be executed, P0 and P1 are deadlocked.

•Another problem related to deadlocks is **indefinite blocking**, or **starvation**, a situation in which processes wait indefinitely within semaphore.

3.6 Classic Problems of Synchronization

1) Bounded-Buffer Problem
2) Readers and Writers Problem
3) Dining-Philosophers Problem

3.6.1 The Bounded-Buffer Problem

•The *bounded-buffer problem* is commonly used to illustrate the power of synchronization primitives.

• The bounded-buffer problem is related to the producer consumer problem.

•We assume that the pool consists of *n* buffers, each capable of holding one item.

•The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.

•The empty and full semaphores count the number of empty and full buffers.

•The semaphore empty is initialized to the value n;

•the semaphore f u l l is initialized to the value 0.

•The code for the producer process is as shown in Figure below:

```
do {
      . . .
   // produce an item in nextp
      . . .
   wait(empty);
   wait(mutex);
      . . .
   // add nextp to buffer
      . . .
   signal(mutex);
   signal(full);
 }while (TRUE),-
```

• The code for the consumer process is as shown in Figure below:.

```
do {
   wait(full);
   wait(mutex);
      . . .
   // remove an item from buffer to nextc
      . . .
   signal(mutex);
   signal(empty);
      . . .
   // consume the item in nextc
      . . .
 }while (TRUE);
```

• The symmetry between the producer and the consumer.
  ¤ The producer produces full buffers for the consumer.
  ¤ The consumer produces empty buffers for the producer.

3.6.2 The Readers-Writers Problem

• A database is to be shared among several concurrent processes.
• Some of these processes may want only to read the database called as readers,
• Some Processes may want to update (that is, to read and write) the database and these are called writers.
• if two readers access the shared data simultaneously, no adverse affects will result.
• but, if a writer and some other thread (either a reader or a writer) access the database simultaneously, it results in data inconsistency
• To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.
• This synchronization problem is referred to as the *readers-writers problem.*
• The reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

• The semaphores mutex and wrt are initialized to 1;
• readcount is initialized to 0.
• The semaphore wrt is common to both reader and writer processes.
• The semaphore wrt functions as a mutual-exclusion semaphore for the writers.
• The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
• The readcount variable keeps track of how many processes are currently reading the object.
• It is also used by the first or last reader that enters or exits the critical section.
• It is not used by readers who enter or exit while other readers are in their critical sections

• The code for a writer process is shown in Figure below

```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
}while (TRUE);
```

- if a writer is in the critical section and *n* readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex.
-when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

• The code for a reader process is shown in Figure below

```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
}while (TRUE);
```

• The readers-writers problem and its solutions has been generalized to provide reader-writer locks on some systems.
• Acquiring a reader-writer lock requires specifying the mode of the lock: either *read* or *write* access.
• When a process only wishes to read shared data, it requests the reader-wrriter lock in read mode;
• a process wishing to modify the shared data must request the lock in write mode.
• Multiple processes are permitted to concurrently acquire a reader-writer lock in read mode;
• only one process may acquire the lock for writing as exclusive access is required for writers.

• Reader-writer locks are most useful in the following situations:
  • In applications where it is easy to identify which processes only read shared data and which threads only write shared data.
  • In applications that have more readers than writers. This is because reader writer locks generally require more overhead to establish than semaphores or mutual exclusion locks, and the overhead for setting up a reader-writer lock is compensated by the increased concurrency of allowing multiple readers.

3.6.3 The Dining-Philosophers Problem

dining-philosophers problem:

• Consider five philosophers who spend their lives thinking and eating.
• The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
• In the center of the table is a bowl of rice, and the table is laidwith five single chopsticks (as shown in Figure).
• When a philosopher thinks, she does not interact with her colleagues.
• From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
• A philosopher may pick up only one chopstick at a time.
• Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
• When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
• When she is finished eating, she puts down both of her chopsticks and starts thinking again.

• **The *dining-philosophers problem* is considered a classic synchronization problem as it is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner**.

Solution:

• Represent each chopstick with a semaphore.
• A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore;
• She releases her chopsticks by executing the signal() operation on the appropriate semaphores.
• Thus, the shared data are

semaphore chopstick[5];

Where all the elements of chopstick are initialized to 1.
• The structure of philosopher i is shown in Figure.

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);

   signal(chopstick[(i+1) % 5]);

   // think

}while (TRUE);
```

• Although this solution guarantees that no two neighbors are eating simultaneously, it could create a deadlock.
• Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
• All the elements of chopstick will now be equal to 0.
• When each philosopher tries to grab her right chopstick, she will be delayed forever.
Several possible remedies to the deadlock problem are listed :
• Allow at most four philosophers to be sitting simultaneously at the table.
• Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
• Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

3.7 Monitors

Semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place and these sequences do not always occur.

Following illustrate various types of errors that can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem.

1) Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore ―mutex‖ are executed, resulting in the following execution:

```
signal(mutex);
    ...
  critical section
    ...
wait(mutex);
```

-In this situation, several processes may be executing in their critical-sections simultaneously, violating the mutual-exclusion requirement.

2) Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

- In this case, a deadlock will occur.

3) Suppose that a process omits the wait(mutex), or the signal(mutex), or both.
-In this case, either mutual-exclusion is violated or a deadlock will occur.

To deal with such errors, researchers have developed high-level language constructs.
One fundamental high-level synchronization construct - the monitor type.
• It provides a convenient and effective mechanism for process synchronization.

3.7.1 Monitors Usage

• A monitor type presents a set of programmer-defined operations that are provided mutual exclusion within the monitor.

• The monitor type contains

- the declaration of variables whose values define the state of an instance of that type, and
- the bodies of procedures or functions that operate on those variables.

• The syntax of a monitor is shown in Figure.

```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }

         .
         .
         .

   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

• The representation of a monitor type cannot be used directly by the various processes.

• Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.

• The monitor construct ensures that only one process at a time can be active within the monitor.

• A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition:*

condition x, y;

• The only operations that can be invoked on a condition variable are **wait () and signal().**

• The operation **x.wait( )** means that the process invoking this operation is suspended until another process invokes x.signal( ) ;

• The **x. signal( )** operation resumes exactly one suspended process.

• If no process is suspended, then the signal( ) operation has no effect;

• That is, the state of x is the same as if the operation had never been executed .

• But the signal( ) operation associated with semaphores, always affects the state of the semaphore.

Figure below shows Schematic view of a monitor.

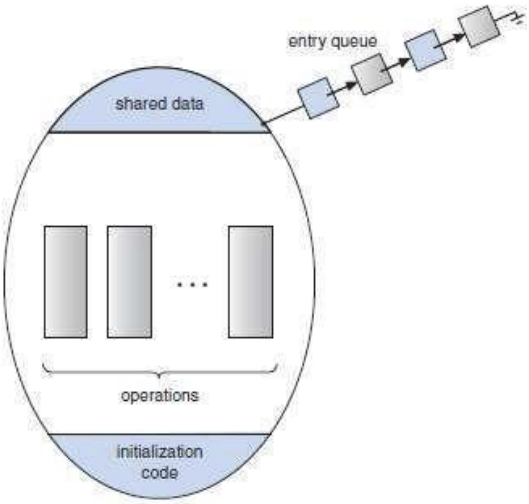
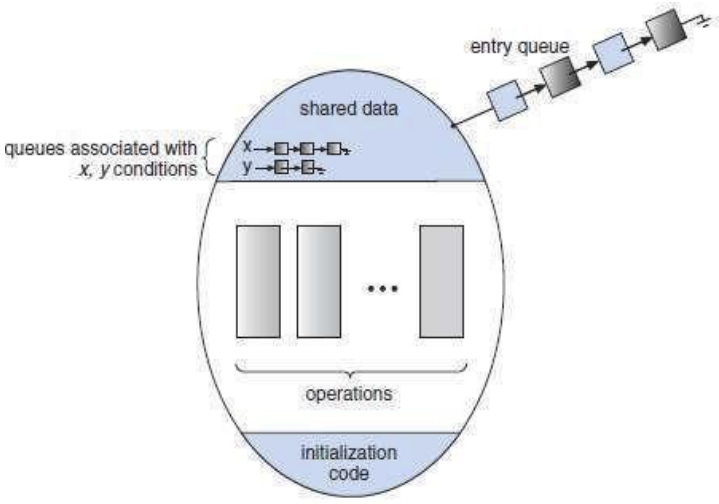


Figure below shows Schematic view of a monitor with condition variab



• Now suppose that, when the x. s ignal () operation is invoked by a process P, there is a suspended process Q associated with condition x.

• if the suspended process Q is allowed to resume its execution, the signaling process P must wait. • Otherwise, both P and Q would be active simultaneously within the monitor.

• But both processes can conceptually continue with their execution.

Two possibilities exist:

1. Signal and wait.

P either waits until Q leaves the monitor or waits for another condition.

2. Signal and continue.

Q either waits until *P* leaves the monitor or waits for another condition.

3.7.2 Dining-Philosophers Solution Using Monitors

• This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

Solution

• three states in which we may find a philosopher:

• therefore the following data structure:

enum {thinking, hungry, eating} state[5] ;

• Philosopher *i* can set the variable state[i] = eating only if her two neighbors are not eating:

(state [(i+4) °/5] != eating) and

(state [(i+1) % 5] != eating).

• We also need to declare condition self [5];

• where philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

• The distribution of the chopsticks is controlled by the monitor dp, whose definition is shown in Figure

• Each philosopher, before starting to eat, must invoke the operation pickup( ).

• This may result in the suspension of the philosopher process.

• After the successful completion of the operation, the philosopher may eat.

• Following this, the philosopher invokes the putdown( ) operation.

• Thus, philosopher *i* must invoke the operations pickup( ) and putdown( ) in the following sequence:

dp.pickup(i);
eat
dp.putdown(i);

• This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state [5];
   condition self [5] ;

   void pickup(int i) {
      state [i] = HUNGRY;
      test (i) ;
      if (state[i] != EATING)
         self [i] .wait();
   }

   void putdown(int i) {
      state til = THINKING;
      test((i + 4) % 5) ;
      test( (i + 1) % 5) ;
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
          state [i] = EATING;
          self [i] .signal();
      }
   }

   initialization-code () {
      for (int i = 0; i < 5; i++)
         state [i] = THINKING;
    }
}
```

3.7.3 Implementing a Monitor using Semaphores

• Each monitor, a semaphore mutex (initialized to 1) is provided

• A process

→ must execute wait(mutex) before entering the monitor and

→ must execute signal(mutex) after leaving the monitor.

• A signaling process must wait until the resumed process either leaves or waits,

• So an additional semaphore, next, is introduced, initialized to 0, on which the signaling processes may suspend themselves.

• An integer variable next-count is also provided to count the number of processes suspended on next.

•
```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

Thus, each external procedure F is replaced by

• Mutual-exclusion within a monitor is ensured.

• Condition variables are implemented as follows

• For each condition variable x, we have:

-semaphore x-sem; (initially = 0)

- integer variable x-count (initialized to 0)

• Definition of x.wait()

```
x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
x_count--;
```

• Definition of x.signal()

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

3.7.4 Resuming Processes within a Monitor

• Problem:

If several processes are suspended, then how to determine which of the suspended processes should be resumed next?

Solution-1: Use an FCFS ordering i.e. the process that has been waiting the longest is resumed first.

Solution-2: Use conditional-wait construct i.e. x.wait(c)

- c is a integer expression evaluated when the wait operation is executed.
- Value of c (a priority number) is then stored with the name of the process that is suspended.
- When x.signal is executed, process with smallest associated priority number is resumed next.

To illustrate this new mechanism, we consider the ResourceAllocator monitor shown in Figure in figure below,

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = TRUE;
   }

   void release() {
      busy = FALSE;
      x.signal();
   }

   initialization_code() {
      busy = FALSE;
   }
}
```

• It controls the allocation of a single resource among competing processes.
• Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource.
• The monitor allocates the resource to the process that has the shortest time allocation request.
• A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    ...
  access the resource;
    ...
R.release();
```

where R is an instance of type ResourceAllocator.

• the monitor concept cannot guarantee that the preceding access sequence will be observed, because of the following problems that can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requestecj.
- A process might request the same resource twice (without first releasing the resource).

• One possible solution to the current problem is to include the resource access operations within the ResourceAllocator monitor.
• However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.
• To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource.
• We must check two conditions to establish the correctness of this system.

- First, user processes must always make their calls on the monitor in a correct sequence.
- Second, we must be sure that an uncooperative process does not simply ignore the mutual exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols.

•Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated.

DEADLOCKS

3.8 System Model

- Every Process needs resources to perform its tasks.
- Every Process will do the following w.r.t resource:-
    1)Request the Resource
    2)Use the Resource
    3)Release the Resource
- There are 2 types of Resources:-
    1)Physical Resources [Memory, CPU, Printer,…..]
    2)Logical Resources [Files, Semaphores,….]
- There can be multiples instances of each resource type [4-printers, 3-scanners]
- Resources can be classified into 2 types:-
    1)Sharable Resources:- Reading a File [n-processes can use [read] the file at a time]
    2)Non Sharable Resources:- Writing a File [Only 1-process is allowed at a time]
- When process requests a resource, the following 2 possibilities may happen:-
    1)If Resource is Free - Resource will be allocated to the requested process
    2)If Resource is Busy - Process will wait until resource will be free [device queue]

Deadlock:-

A set of processes {P1, P2, …..,Pn} are in deadlock state, if and only if every process in the set is waiting for a resource that is held by another process in the set.

- Waiting (vs) Deadlock ?
    - Deadlock -Indefinite waiting
    - Processes won't proceed further even after 1000 years.

3.9 Deadlock Characterization

- In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

3.9.1 Necessary Conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion.
    -At least one resource must be held in a non-sharable mode;
    -only one process at a time can use the resource.
    -If another process requests that resource, the requesting process must be delayed until the resource has been released.

2) Hold and Wait

A process must be simultaneously

→ holding at least one resource and

→ waiting to acquire additional resources held by the other process.

3) No Preemption

Resources cannot be preempted.; a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4) Circular Wait

A set of processes { P0, P1, P2, . . ., PN } waiting must exist such that P0 is waiting for a resource that is held by P1 ,P1 is waiting for a resource that is held by P2, ….Pn-1 is waiting for a resource held by Pn, and Pn, is waiting for a resource held by P1.

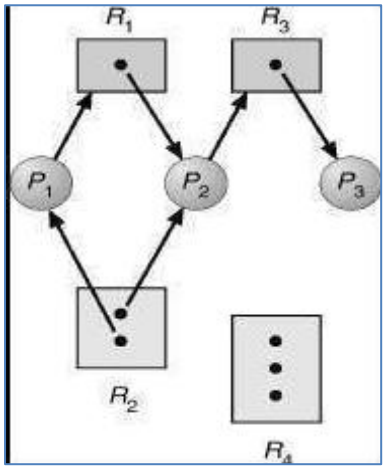
• All four conditions must hold for a deadlock to occur.
• The four conditions are not completely independent.

3.9.2 Resource-Allocation-Graph

• The resource-allocation-graph (RAG) is a directed graph that can be used to describe the deadlock situation.
• RAG consists of a
→ set of vertices (V) and
→ set of edges (E).
• V is divided into two types of nodes
1) P={P1,P2……..Pn} i.e., set consisting of all active processes in the system.
2) R={R1,R2…….Rn} i.e., set consisting of all resource types in the system.
• E is divided into two types of edges:
1) Request Edge
- A directed-edge $P_i \rightarrow R_j$ is called a request edge.
- $P_i \rightarrow R_j$ indicates that process P_i has requested a resource R_j.
2) Assignment Edge
- A directed-edge $R_j \rightarrow P_i$ is called an assignment edge.
- $R_j \rightarrow P_i$ indicates that a resource R_j has been allocated to process P_i.
• Pictorially,
→ each process P_i is represented as a circle.
→ each resource-type R_j is represented as a rectangle.
→resource type *Rj* may have more than one instance, each such instance is represented as a dot within the rectangle.
→a request edge points to only the rectangle *R*
→an assignment edge must also designate one of the dots in the rectangle.

• When process P, requests an instance of resource type *Rj,* a request edge is inserted in the resource-allocation graph.
• When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge.
• When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted

• The resource-allocation graph shown in Figure depicts the following situation.



• The sets P, R, and E:

P={P1 ,P2, P3,}

R= {R1, R2, R3, R4}

E = {P1→ R1, P2→R3,R1→P2,R2→P2,R2→P1,R3→P3 }

• Resource instances:

-One instance of resource type R1

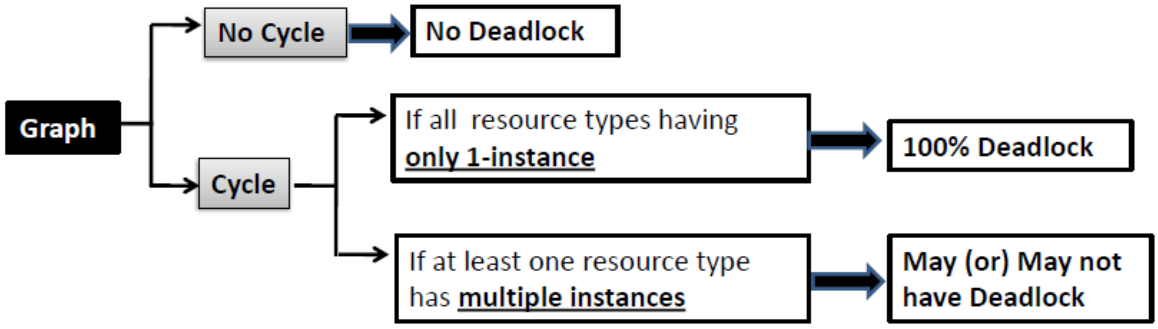
-Two instances of resource type R2

-One instance of resource type R3

-Three instances of resource type R4

• Process states:

- Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
- Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
- Process P3 is holding an instance of R3.

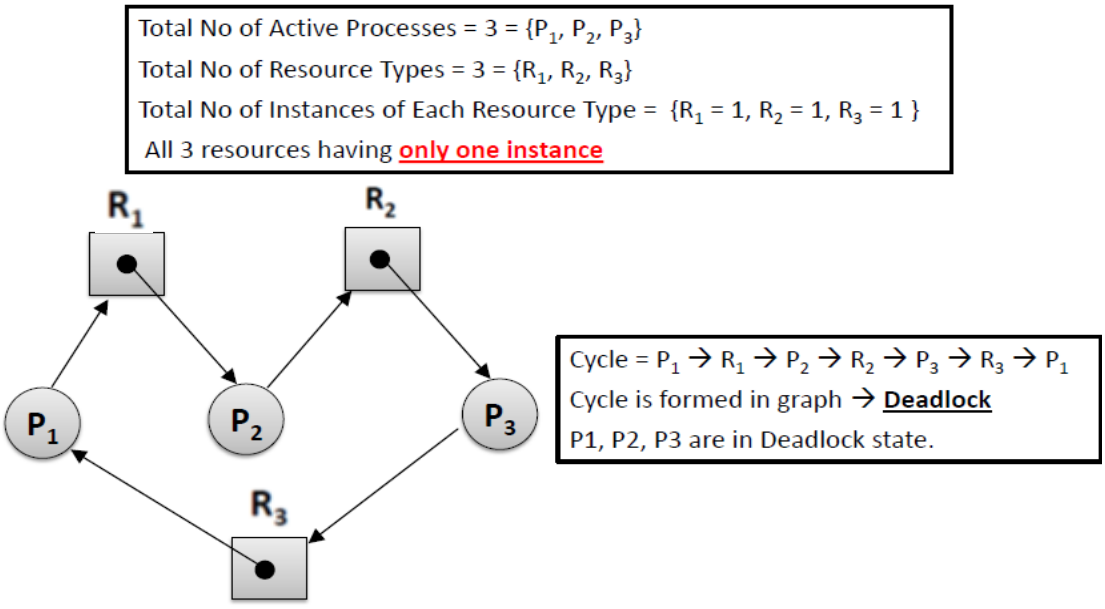


Case I :Cycle with Deadlock with all Resource Types having only 1-instance.

Case II :Cycle with Deadlock with at least one Resource type having multiple instances.

Case III :Cycle with No Deadlock with at least one Resource type having multiple instances.

Case I :Cycle with Deadlock with all Resource Types having only 1-instance.


Total No of Active Processes = 3 = {P_1, P_2, P_3}
Total No of Resource Types = 3 = {R_1, R_2, R_3}
Total No of Instances of Each Resource Type = {R_1 = 1, R_2 = 1, R_3 = 1 }
All 3 resources having only one instance
R_1
R_2
P_1
P_2
P_3
R_3
Cycle = P_1 → R_1 → P_2 → R_2 → P_3 → R_3 → P_1
Cycle is formed in graph → Deadlock
P1, P2, P3 are in Deadlock state.

Case II :Cycle with Deadlock with at least one Resource type having multiple instances.

• If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.

• In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, consider the resource-allocation graph depicted in Figure(a)

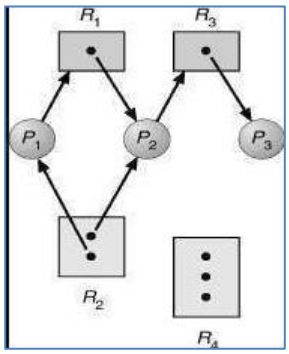


Figure (a)

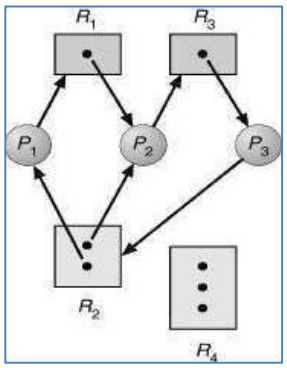


figure (b)

• Suppose that process P3 requests an instance of resource type *R2.*

• Since no resource instance is currently available, a request edge P3 —> R2 is added to the graph as shown Figure(b).

• At this point, two minimal cycles exist in the svstem:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

• Processes *P1, P2,* and P3 are deadlocked.

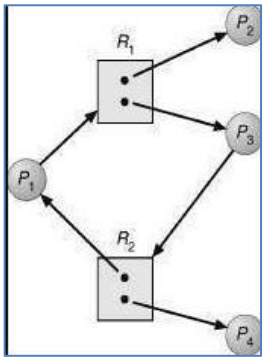
- Process P2 is waiting for the resource R3, which is held by process P3.
- Process P3 is waiting for either process *P1* or process *P2* to release resource R2.
- Process P1 is waiting for process P2 to release resource R1.

Case III :Cycle with No Deadlock with at least one Resource type having multiple instances.

• consider the resource-allocation graph in Figure

• In this example, we also have a cycle

P1 → R1 → P3→ R2 → P1



• However, there is no deadlock.

• Observe that process P4 may release its instance of resource type R2.

• That resource can then be allocated to P3, breaking the cycle

In summary

- if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state.

-If there is a cycle, then the system may or may not be in a deadlocked state.

3.10 Methods for Handling Deadlocks

The following are the 3 ways for handling deadlocks:-

1)Ensure that system will never enter into a deadlock state by

1.Deadlock Prevention

-Deadlock prevention provides a set of methods for ensuring that at least one of the necessary conditions cannot hold.

2. Deadlock Avoidance

-Deadlock avoidance requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime.

-With this additional knowledge, it can decide for each request whether or not the process should wait.

2) Allow the system to enter into deadlock state, detect it & recover from it by

1.Deadlock Detection

-an algorithm that examines the state of the system to determine whether a deadlock has occurred

2. Recovery from Deadlock

- an algorithm to recover from the deadlock

3) Ignore the problem altogether and pretend that deadlocks never occur in the system

-In this case, the undetected deadlock will result in deterioration of the system's performance

-Eventually, the system will stop functioning and will need to be restarted manually.

-this method is used in most operating systems,

In many systems, deadlocks occur infrequently (say, once per year);

-thus, this method is cheaper than the prevention, avoidance, or detection and recovery methods

3.11 Deadlock-Prevention

• Deadlocks can be eliminated by preventing at least one of the four required conditions:

1) Mutual exclusion
2) Hold-and-wait
3) No preemption
4) Circular-wait.

3.11.1 Mutual Exclusion

• This condition must hold for non-sharable resources.
• For example: A printer cannot be simultaneously shared by several processes.
• On the other hand, shared resources do not lead to deadlocks.
• For example: Simultaneous access can be granted for read-only file.
• A process never waits for accessing a sharable resource.
• In general, we cannot prevent deadlocks by denying the mutual-exclusion condition because some resources are non-sharable by default.

3.11.2 Hold and Wait

• To prevent this condition:

The processes must be prevented from holding one or more resources while simultaneously waiting for one or more other resources.

• Two possible solution to this problem.
• For example:

Consider a process that

→ copies the data from a tape drive to the disk
→ sorts the file and
→ then prints the results to a printer.

Protocol-1

Each process must be allocated with all of its resources before it begins execution.
-All the resources (tape drive, disk files and printer) are allocated to the process at the beginning.

Protocol-2

A process must request a resource only when the process has none.
- Initially, the process is allocated with tape drive and disk file.
- The process performs the required operation and releases both tape drive and disk file.
- Then, the process is again allocated with disk file and the printer
- the process performs the required operation & releases both disk file and the printer.

• Disadvantages of above 2 methods:

1) Resource utilization may be low, since resources may be allocated but unused for a long period.
2) Starvation is possible.
A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3.11.3 No Preemption

• To prevent this condition: the resources must be preempted.
• There are several solutions to this problem.

Protocol-1

• If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources currently being held are preempted.
• The preempted resources are added to the list of resources for which the process is waiting.
• The process will be restarted only when it regains the old resources and the new resources that it is requesting.

Protocol-2

• When a process request resources, we check whether they are available or not.
• If they are, we allocate them.
• If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
• If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
• If the resources are neither available nor held by a waiting process, the requesting process must wait.
• While it is waiting, some of its resources may be preempted, but only if another process requests them.
• A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.

• These 2 protocols may:
- be applicable for resources whose states are easily saved and restored, such as registers and memory.
- not applicable to other devices such as printers and tape drives.

3.11.4 Circular-Wait

• One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

• Each resource type a unique integer number, which, allows us to compare two resources and to determine whether one precedes another in our ordering.

• we define a one-to-one function F: R —> N, where N is theset of natural numbers.

i.e F(R1) = 1
F(R2) = 2
F(R3) = 3

• Deadlock can be prevented by using the following 2 protocol:

Protocol :1

- Require the processes to request resources only in increasing/decreasing order.

Protocol-2

-If process wants to request "Rj", then it must release all allocated resources "Ri" where F(Ri) > F(Rj). [R2, R3 to be released to request R1].

• One big challenge in this scheme is determining the relative ordering of the different resources.

3.12 Deadlock Avoidance

• Deadlock-prevention algorithms, prevent deadlocks by restraining how requests can be made.

• The restraints ensure that at least one of the necessary conditions for deadlock cannot occur and, hence, that deadlocks cannot hold.

• Possible side effects of preventing deadlocks : low device utilization and reduced system throughput.

• The general idea behind deadlock avoidance is to avoid deadlocks from ever happening.

• Deadlock-avoidance algorithm

→ requires more information about each process.

• With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

• Each request requires in making the decision based on

→ the resources currently available,
→the resources currently allocated to each process, and
→the future requests and releases of each process.

• Various algorithms that use this approach differ in the amount and type of information required.

1)In simple algorithms, the algorithm only needs to know the maximum number of each resource that a process might potentially use.

2) In complex algorithms, the algorithm can also take advantage of the schedule of exactly what resources may be needed in what order.

• The resource-allocation state is defined by

→ the number of available resources
→allocated resources and
→ the maximum demand of each process.

• A deadlock-avoidance algorithm dynamically examines the resources allocation state to ensure that a circular-wait condition never exists.

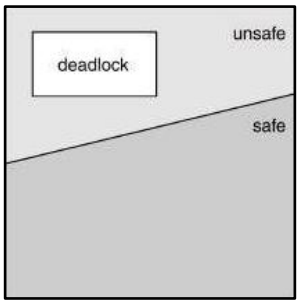
3.12.1 Safe State

• A state is safe if the system can allocate all resources requested by all processes without entering a deadlock state.

• A state is safe if there exists a safe sequence of processes {P0, P1, P2, ..., PN} such that the requests of each process(Pi) can be satisfied by the currently available resources.

• If a safe sequence does not exist, then the system is in an unsafe state, which may lead to deadlock.

• All safe states are deadlock free, but not all unsafe states lead to deadlocks as shown in figure.



• To illustrate, we consider a system with 12 magnetic tape drives and three processes: P0, *P1,* and P3.

• Table below shows maximum resource needed , current allocation and how much they need to complete the work.

Process	Maximum need	Current allocation	Need
P0	10	5	5
P1	4	2	2
P2	9	2	7

• Thus, there are 3 free tape drives.

• At time t0, the system is in a safe state.

• The sequence < P1, *P0, P2*> satisfies the safety condition.

• Since 2 tape drive is left ,Process P1 can immediately be allocated all its tape drives, and then return them

• Therefore the system will then have 5 available tape drives);

• Then process P0 can get all its tape drives and return them

• the system will then have10 available tape drives and

• finally process P2 can get all its tape drives and return them

• thus the system will then have all 12 tape drives available.

At time t2, process *P2* requests one more tape drive and is allocated one more tape drive, then table shows current state,

process	Maximum need	Current allocation	Need
P0	10	5	5
P1	4	2	2
P2	9	3	6

• Thus, there are 2 free tape drives left.

• Now process P1 can complete its work as it needs only 2 tape drive, thus returns them

• Now System have 4 tape drives

• Neither P0 nor P2 can complete as P0 require 5 tape drives and P2 requires 6 tape drives.

• A system can go from a safe state to an unsafe state.

• resulting in a deadlock.

• mistake was in granting the request from process *P2* for one more tape drive.

If all resource types having **only 1**-instance → **Use Resource Allocation Graph**

If at least one resource having **multiple** instances → **Use Bankers Algorithm**

3.12.2 Resource-Allocation-Graph Algorithm

• If resource categories have only single instances of their resources, then deadlock states can be detected by cycles in the resource-allocation graphs.

• In this case, unsafe states can be recognized and avoided by augmenting the resource-allocation graph with claim edges (denoted by a dashed line).

• Claim edge Pi → Rj indicated that process Pi may request resource Rj at some time in future.

• The important steps are as below:

1) When a process Pi requests a resource Rj, the claim edge Pi → Rj is converted to a request edge.

2) Similarly, when a resource Rj is released by the process Pi, the assignment edge Rj → Pi is reconverted as claim edge Pi → Rj.

3) The request for Rj from Pi can be granted only if the converting request edge to assignment edge do not form a cycle in the resource allocation graph.

• To apply this algorithm, each process Pi must know all its claims before it starts executing.

• Conclusion:

1) If no cycle exists, then the allocation of the resource will leave the system in a safe state.

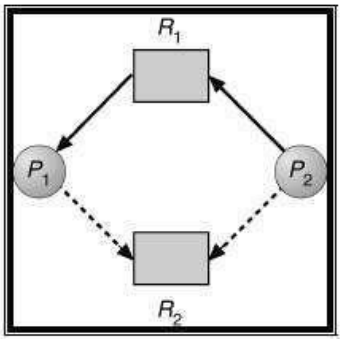
2) If cycle is found, system is put into unsafe state and may cause a deadlock.

• For example: Consider a resource allocation graph shown in Figure(a).

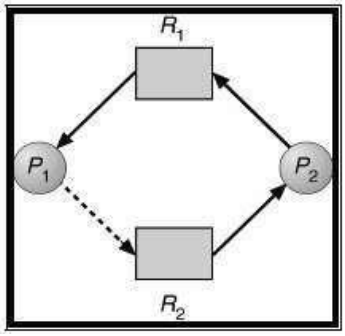
- Suppose P2 requests R2.

- Though R2 is currently free, we cannot allocate it to P2 as this action will create a cycle in the graph as shown in Figure (b).

- This cycle will indicate that the system is in unsafe state: because, if P1 requests R2 and P2 requests R1 later, a deadlock will occur.



(a) For deadlock avoidance



(b) an unsafe state

• Problem:

The resource-allocation graph algorithm is not applicable when there are multiple instances for each resource.

• Solution:

Use banker's algorithm.

3.12.3 Banker's Algorithm

• This algorithm is applicable to the system with multiple instances of each resource types.
• However, this algorithm is less efficient than the resource-allocation-graph algorithm.
• When a process starts up, it must declare the maximum number of resources that it may need.
• This number may not exceed the total number of resources in the system.
• When a request is made, the system determines whether granting the request would leave the system in a safe state.
• If the system in a safe state,

the resources are allocated;

else

the process must wait until some other process releases enough resources.

• Assumptions:

Let n = number of processes in the system

Let m = number of resources types.

• Following data structures are used to implement the banker's algorithm.

1) Available:

- A vector of length m, indicates the no. of available resources of each type.
- If Available[j]=k, then k instances of resource type Rj is available.

2) Max

- an n X m matrix indicates the maximum demand of each process of each resource.
- If Max[i][j]=k, then process Pi may request at most k instances of resource type Rj.

3) Allocation

- an n X m matrix ,indicates no. of resources currently allocated to each process.
- If Allocation[i][j]=k, then Pi is currently allocated k instances of Rj.

4) Need

- an n X m matrix, indicates the remaining resources need of each process.
-If Need[i][j]=k, then Pi may need k more instances of resource Rj to complete its task.
- So, Need[i][j] = Max[i][j] - Allocation[i][j]

Notation:

•Let X and Y be vectors of length n.
•We say that X < Y if and only if X[i] < Y[i] for all i = 1, 2, ..., n and X !=Y.
•For example, if X= (1,7,3,2) and Y = (0,3,2,1), then Y <= X.,
• each row in the matrices Allocation and Need as vectors refered as $Allocation_i$ and $Need_i$.
•The vector $Allocation_i$, specifies the resources currently allocated to process Pi;
• the vector $Need_i$ specifies the additional resources that process Pi may still request to complete its task.

• The Banker's algorithm has two parts: 1) Safety Algorithm
2) Resource - Request Algorithm

3.12.3.1 Safety Algorithm

Algorithm to verify whether the system is safe (or) not after allocating the requested resource

1. Work[m] = Available → Work contains the current availability
   Finish[n] = False for i = 0, 1, 2,...,n-1 → Assume that all processes are false initially
2. Find an index "i" such that both
   a) Finish[i] = false
   b) $Need_i \leq work$ → Need of Process Pi < Availability
   If no such "i" exists, go to step 4.
3. Work = Work + $Allocation_i$ → "Pi" got all resources finished & released all resources
   Finish[i] = True → Process "Pi" finished its execution
   go to step2.
4. If Finish[i] = True for all i = 0, 1, 2,...,n-1 → System is in Safe State

3.12.3.2 Resource-Request Algorithm

- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one.
- If so, grant the request, and if not, deny the request.
- Let $Request_i$ be the request vector of process Pi.
- If $Request_i[j]=k$, then process Pi wants K instances of the resource type Rj.
- When a request for resources is made by process *Pi*, the following actions are taken:

1. If **$Request_i \leq Need_i$**, got to **step2**.
   Otherwise raise an error , since process "Pi" has exceeded its maximum claim.
2. If **$Request_i \leq Available_i$**, got to **step3**.
   Otherwise process "Pi" must wait, Since resources are not available.
3. Now **allocate** the requested resources to process "Pi" from the available resources.
   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;
4. Now **run** the **Safety Algorithm.**
   If **Safe** → Transaction is completed [Resources has been allocated]
   If **Unsafe** → Old-Resource allocation state is restored and process "Pi" waits.

3.12.3.3 An Illustrative Example

Question: Consider the following snapshot of a system:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	3	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

- The content of the matrix Need
  - given by Need = Max - Allocation
  - So, the content of Need Matrix is:

	Need		
	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

- Applying the Safety algorithm on the given system,

Step 1: Initialization

Work = Available i.e. Work =3 3 2

…P0……P1…P2…..P3……P4.

Finish = | false | false | false | false | false |

Step 2: For i=0 , Need[P0] > Work, so next For i=1

Finish[P1] = false and Need[P1]<=Work i.e. (1 2 2)<=(3 3 2) - true

So P1 can complete its task and hence must be kept in safe sequence.

Step 3: Work = Work + Allocation[P1] =(3 3 2)+(2 0 0)=(5 3 2)

…….P0…P1…….P2….P3…..P4……

Finish = | false | true | false | false | false |

Step 2: For i=2

Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0)<=(5 3 2) - false

So P2 must wait.

Step 2: For i=3

Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) - true

So P3 can complete its task and hence must be kept in safe sequence.

Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3)

……P0…P1…P2…..P3….P4….

Finish = | false | true | false | true | false |

Step 2: For i=4

Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) - true

So P4 can complete its task and hence must be kept in safe sequence.

Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5)

…P0……P1… P2… P3….P4…..

Finish= | false | true | false | true | true |

Step 2: For i=0

Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) -true

So P0 can complete its task and hence must be kept in safe sequence.

Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5)
…..P0….P1…P2….P3….P4….
Finish= | true | true | false | true | true |

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) - true
So P2 can complete its task and hence must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7)
..P0… .P1……P2….P3…P4….
Finish= | true | true | true | true | true |

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is currently in a safe state.
The safe sequence is <P1, P3, P4, P0, P2>.

Conclusion: Yes, the system is currently in a safe state.

• If a request from process P1 arrives for (1 0 2) can the request be granted immediately:

• To decide whether the request is granted, we use Resource Request algorithm.

Step 1: Request[P1]<=Need[P1] i.e. (1 0 2)<=(1 2 2) - true.

Step 2: Request[P1]<=Available i.e. (1 0 2)<=(3 3 2) - true.

Step 3: Available = Available - Request[P1] = (3 3 2) - (1 0 2)= (2 3 0)

Allocation[P1] = Allocation[P1] + Request[P1] = (2 0 0) + (1 0 2)= (3 0 2)

Need[P1] = Need[P1] - Request[P1] = (1 2 2) - (1 0 2)= (0 2 0)

• We arrive at the following new system state:

	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	2	3	0
P1	3	0	2	3	2	2			
P2	3	0	3	9	0	2			
P3	2	1	1	2	2	2			
P4	0	0	2	4	3	3			

• The content of the matrix Need is given by

Need = Max - Allocation

• So, the content of Need matrix is:

	Need		
	A	B	C
P0	7	4	3
P1	0	2	0
P2	6	0	0
P3	0	1	1
P4	4	3	1

• To determine whether this new system state is safe, we again execute Safety algorithm.

Step 1: Initialization
Here, m=3, n=5
Work = Available i.e. Work =2 3 0
….P0……P1……P2……P3…….P4….
Finish = | false | false | false | false | false |

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(2 3 0) · false
So P0 must wait.

Step 2: For i=1
Finish[P1] = false and Need[P1]<=Work i.e. (0 2 0)<=(2 3 0) · true
So P1 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P1] =(2 3 0)+(3 0 2)=(5 3 2)

	P0	P1	P2	P3	P4
Finish =	false	true	false	false	false

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(5 3 2) · false
So P2 must wait.

Step 2: For i=3
Finish[P3] = false and Need[P3]<=Work i.e. (0 1 1)<=(5 3 2) · true
So P3 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P3] = (5 3 2)+(2 1 1)=(7 4 3)

	P0	P1	P2	P3	P4
Finish =	false	true	false	true	false

Step 2: For i=4
Finish[P4] = false and Need[P4]<=Work i.e. (4 3 1)<=(7 4 3) · true
So P4 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P4] =(7 4 3)+(0 0 2)=(7 4 5)

	P0	P1	P2	P3	P4
Finish =	false	true	false	true	true

Step 2: For i=0
Finish[P0] = false and Need[P0]<=Work i.e. (7 4 3)<=(7 4 5) · true
So P0 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P0] =(7 4 5)+(0 1 0)=(7 5 5)

	P0	P1	P2	P3	P4
Finish =	true	true	false	true	true

Step 2: For i=2
Finish[P2] = false and Need[P2]<=Work i.e. (6 0 0) <=(7 5 5) · true
So P2 must be kept in safe sequence.
Step 3: Work = Work + Allocation[P2] =(7 5 5)+(3 0 2)=(10 5 7)

	P0	P1	P2	P3	P4
Finish=	true	true	true	true	true

Step 4: Finish[Pi] = true for 0<=i<=4
Hence, the system is in a safe state.
The safe sequence is <P1, P3, P4, P0, P2>.

Conclusion: Since the system is in safe sate, the request can be granted.

3.13 Deadlock Detection

• If a system does not employ either deadlock-prevention or deadlock-avoidance algorithm then a deadlock may occur.

• In this environment, the system must provide

1) An algorithm to examine the system-state to determine whether a deadlock has occurred.

2) An algorithm to recover from the deadlock.

3.13.1 Single Instance of Each Resource Type

• If all the resources have only a single instance, then deadlock detection-algorithm can be defined using a wait-for-graph.

• A wait-for-graph (WAG) is a variation of the resource-allocation-graph.

• The wait-for-graph is applicable only to a resource type with single instance.

• The wait-for-graph can be obtained from the resource-allocation-graph by

→ removing the resource nodes and

→ collapsing the appropriate edges.

• An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.

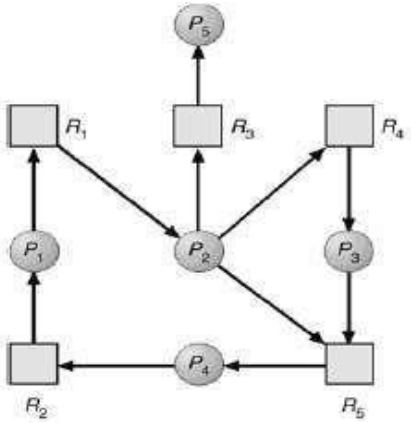
• An edge $P_i \rightarrow P_j$ exists if and only if the corresponding graph contains two edges

1) $P_i \rightarrow R_q$ and

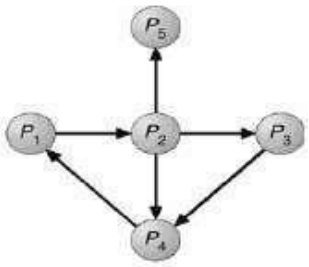
2) $R_q \rightarrow P_j$.

• For example:

Consider resource-allocation-graph shown in Figure (a) and its corresponding wait-for-graph is shown in Figure(b).



Figure(a) Resource-allocation-graph



Figure(b) wait-for-graph.

• A deadlock exists in the system if and only if the wait-for-graph contains a cycle.

• To detect deadlocks, the system needs to

→ maintain the wait-for-graph and

→ periodically execute an algorithm that searches for a cycle in the graph.

3.13.2 Several Instances of a Resource Type

• The wait-for-graph is applicable to only a single instance of a resource type.
• Problem: However, the wait-for-graph is not applicable to a multiple instance of a resource type.
• Solution: The following detection-algorithm can be used for a multiple instance of a resource type.
• Assumptions:

Let 'n' be the number of processes in the system
Let 'm' be the number of resources types.

• Following data structures are used to implement this algorithm.

1) Available
- A vector of length m, indicates the no. of available resources of each type.
- If Available[j]=k, then k instances of resource type Rj is available.

2) Allocation
-A n X m matrix indicates no. of resources currently allocated to each process.
- If Allocation[i][j]=k, then Pi is currently allocated k instances of Rj.

3) Request
- A n X m matrix indicates the current request of each process.
-If Request [i][j] = k, then process Pi is requesting k more instances of resource type Rj.

Algorithm with at least 1-resource type having multiple instances

1. Work[m] = Available [Work contains the current Availability]
   Finish[n] = False if **$Allocation_i$!=0** for all i = 0, 1, 2,....., n-1
   Finish[n] = True if $Allocation_i$ =0 for all i = 0, 1, 2,....., n-1

2. Find an index "i" such that both
   $Request_i \leq work$
   If no such "i" exists, go to step 4.

3. Work = Work + $Allocation_i$ → "Pi" got all resources finished & released all resources
   Finish[i] = True → Process "Pi" finished its execution
   go to step2.

4. If **Finish[i] = False** for some "i" where $0 \leq i < n$, then the system is in deadlocked state.
   Moreover, if Finish[i] = false → Process "Pi" is involved in deadlock.

• To illustrate this algorithm,
• we consider a system with five processes *P0* through P4 and three resource types *A, B,* and C. Resource type *A* has seven instances, resource type B has two instances, and resource type C has six instances.
• Suppose that, at time To, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*ABC*	*ABC*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

• the system is not in a deadlocked state.
• if we execute our algorithm, we will find that the sequence <P0, *P2, P3, P1, P4*> results in *Finish[i] == true* for all *i.*

• Suppose now that process *P2* makes one additional request for an instance of type C.
• The *Request* matrix is modified as follows:

	Request
	A BC
P_0	0 00
P_1	2 02
P_2	0 01
P_3	1 00
P_4	0 02

• The system is now deadlocked.
• Although we can reclaim the resources held by process Po, the number of available resources is not sufficient to fulfill the requests of the other processes.
• Thus, a deadlock exists, consisting of processes *P1,* P2, P3, and P4.

3.13.3 Detection-Algorithm Usage

• The detection-algorithm must be invoked based on following factors:

1. How *often* is a deadlock likely to occur?
2. How *many* processes will be affected by deadlock when it happens?

• If deadlocks occur frequently, then the detection-algorithm should be executed frequently.

• Resources allocated to deadlocked-processes will be idle until the deadlock is broken.

• Problem:

Deadlock occurs only when some processes make a request that cannot be granted immediately.

• Solution 1:

-The deadlock-algorithm must be executed whenever a request for allocation cannot be granted immediately.

-In this case, we can identify

→ set of deadlocked-processes and

→ specific process causing the deadlock.

-Problem: if the deadlock-detection algorithm is invoked for every resource request, this will incur a considerable overhead in computation time.

• Solution 2:

-The deadlock-algorithm must be executed in periodic intervals.

- For example:

→ once in an hour

→ whenever CPU utilization drops below certain threshold

-Problem:

-If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph.

-In this case, we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

3.14 Recovery from deadlock

• approaches to recovery from deadlock:

- Inform the system-operator and let the operator deal with the deadlock manually
- let the system *recover* from the deadlock automatically.

- There are two options for breaking a deadlock.

1) to abort one or more processes to break the circular wait.

2) to preempt some resources from one or more of the deadlocked processes.

3.14.1 Process Termination

• Two methods to remove deadlocks:

1) Abort all deadlocked-processes.

-This method will definitely break the deadlock-cycle.

- However, this method incurs great expense. This is because

→ Deadlocked-processes might have computed for a long time.

→ Results of these partial computations must be discarded.

→ Probably, the results must be re-computed later.

2) Abort one process at a time until the deadlock-cycle is eliminated.
-This method incurs large overhead.
-This is because after each process is aborted,
-deadlock-algorithm must be executed to determine if any other process is still deadlocked

• Aborting a process may not be easy. Many factors may affect which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed and how much longer the process will compute before completing its designated task
3. How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
4. How many more resources the process needs in order to complete
5. How many processes will need to be terminated
6. Whether the process is interactive or batch

3.14.2 Resource Preemption

• Some resources are taken from one or more deadlocked-processes.
• These resources are given to other processes until the deadlock-cycle is broken.
• Three issues need to be considered:

1) Selecting a victim
- Which resources/processes are to be pre-empted (or blocked)?
- The order of pre-emption must be determined to minimize cost.
- Cost factors includes
    1. The time taken by deadlocked-process for computation.
    2. The no. of resources used by deadlocked-process.

2) Rollback
- If a resource is taken from a process, the process cannot continue its normal execution.
- In this case, the process must be rolled-back to some safe state
-This method requires the system to keep more information about the state of all running processes.
- it is difficult to determine what a safe state is,
- the simplest solution is a total rollback: Abort the process and then restart it.

3) Starvation
Problem: In a system where victim-selection is based on cost-factors, the same process may be always picked as a victim.
-As a result, this process never completes its designated task, leading to starvation.
Solution:
- Ensure a process is picked as a victim only a (small) finite number of times.
- example: based on number of rollbacks.

MODULE 4

MEMORY MANAGEMENT

Memory management strategies:

4.1 Background

• Memory management is concerned with managing the primary memory.
• Memory consists of array of bytes or words each with their own address.
• The instructions are fetched from the memory by the cpu based on the value program counter.

4.1.1 Basic Hardware

• Program must be
→ brought (from disk) into memory and
→ placed within a processor for it to be run.
• Main-memory and registers are only storage CPU can access directly.
• Register can be access in one CPU clock.
• Main-memory can take many cycles.
•hence Cache a memory buffer,which sits between main-memory and CPU registers.

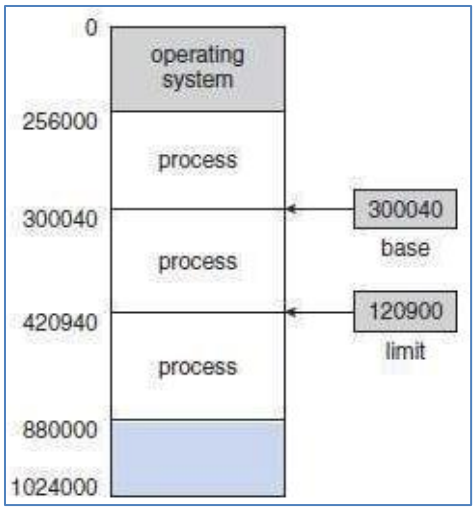


figure (a)

• we also must ensure correct operation, has to protect the operating system from access by user processes and, in addition, to protect user processes from one another.
• This protection must be provided by the hardware.
• each process has a separate memory space.
• and also has the range of legal addresses that the process may access and the process can access only these legal addresses.
• this protection is provided by using two registers, a base and a limit,
• The base register holds the smallest legal physical memory address;
• the limit register specifies the size of the range.

• For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (300040+120900) (inclusive). illustrated in Figure (a)

• Protection of memory space is accomplished by having the CPU hardware compare *every* address generated in user mode with the registers(base and limit).
• Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system, which treats the attempt as a fatal error (as shown in Figure b).

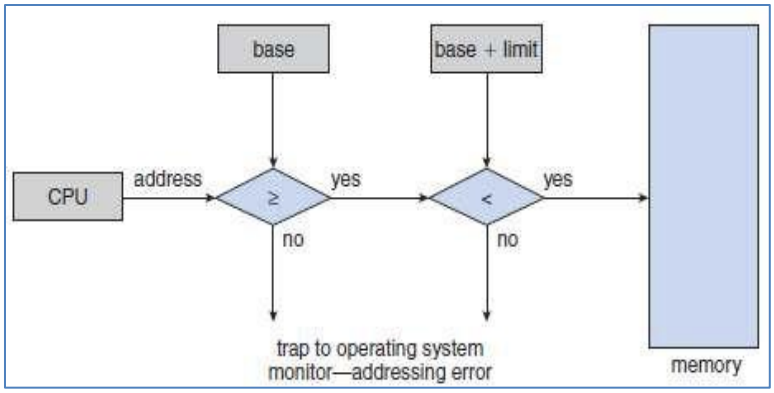
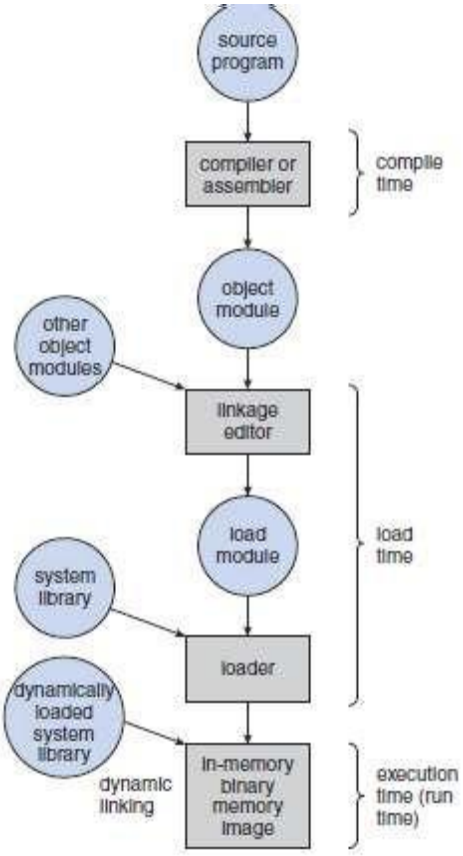


Figure (b)

•This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.
• The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.
• The operating system, executing in kernel mode, is given unrestricted access to both operating system and users' memory.

4.1.2 Address Binding

• A program resides on a disk as a binary executable file.
• To be executed, the program must be brought into memory and placed within a process.
• The processes on the disk that are waiting to be brought into memory for execution form the input queue.
• A user process to reside in any part of the physical memory.
• Although the address space of the computer starts at 00000, the first address of the user process need not be 00000.
• This approach affects the addresses that the user program can use.



Multistep processing of a user program

• A user program will go through several steps—some of which maybe optional-—before being executed (as shown in Figure below).
• Addresses may be represented in different ways during these steps.
• Addresses in the source program are generally symbolic (such as *count).*
• *A* compiler will typically bind these symbolic addresses to relocatable addresses
• The linkage editor or loader will in turn bind the relocatable addresses to absolute addresses
• Each binding is a mapping from one address space to another.

• Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

- Compile time.
  - If you know at compile time where the process will reside in memory, then absolute code can be generated.
  - For example, if you know that a user process will reside starting at location *R,* then the generated compiler code will start at that location and extend up from there.
  - If, at some later time, the starting location changes, then it will be necessary to recompile this code.
- Load time.
  - If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.
  - In this case, final binding is delayed until load time.
  - If the starting address changes, we need only reload the user code to incorporate this changed value.
- Execution time.

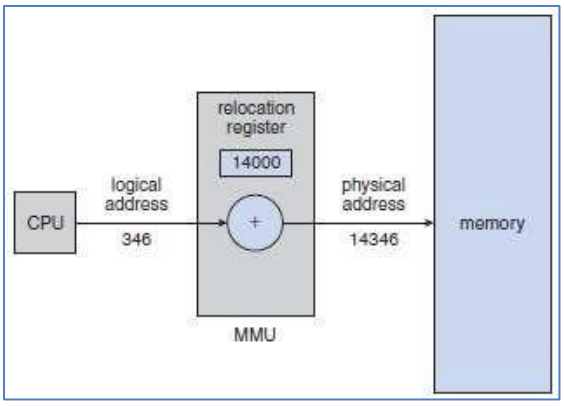
• If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.
• Special hardware must be available for this scheme to work,

4.1.3 Logical versus Physical Address Space

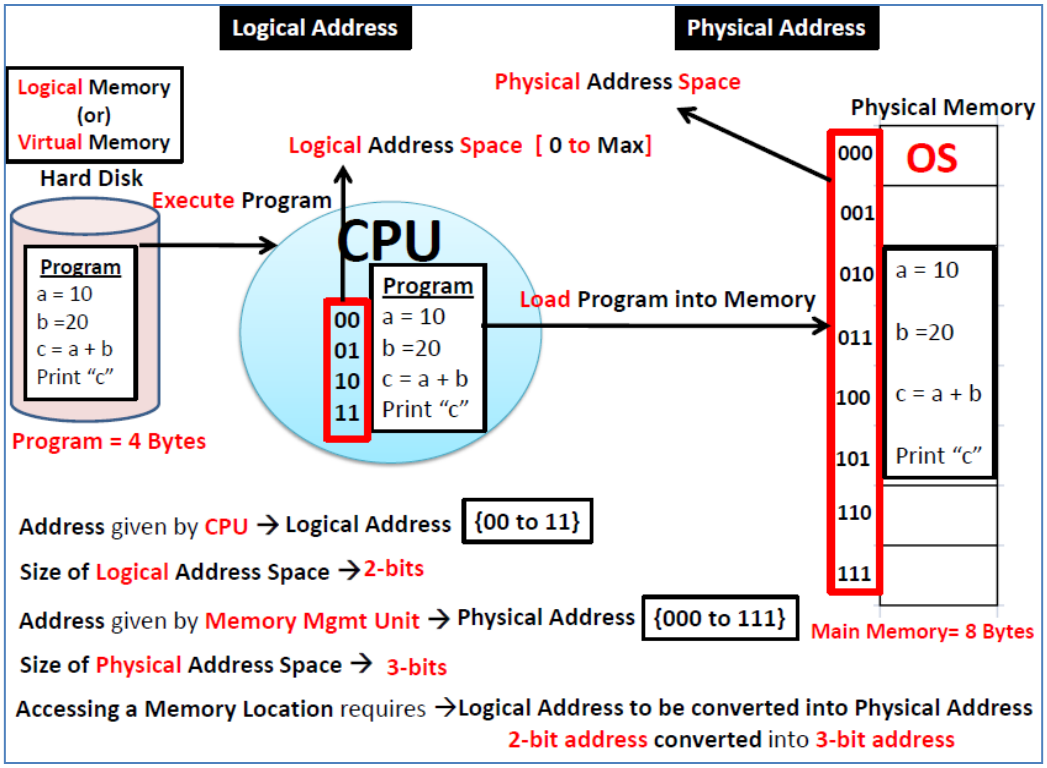
• Logical-address is generated by the CPU (also referred to as virtual-address).
• Physical-address is the address seen by the memory-unit i.e, the one loaded into the MAR(memory address register)

• The set of all logical addresses generated by a program is a logical address space
• The set of all physical addresses corresponding to these logical addresses is a physical address space.

• Logical & physical-addresses are identical in compile-time & load-time address-binding methods.
• Logical and physical-addresses differ in execution-time address-binding method.



• The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).
• The user program never sees the *real* physical addresses.
• The user program deals with *logical* addresses.
• The memory-mapping hardware converts logical addresses into physical addresses
• The base register is now called a relocation register.
• The value in the relocation register is *added* to every address generated by a user process that is logical address at the time it is sent to memory to get the physical address as shown in figure.
• For example, if the base is at 14000, then an attempt by the user to access a location 346 is mapped to location 14346.



4.1.4 Dynamic Loading

• Dynamic Loading can be used to obtain better memory-space utilization.
• A routine is not loaded until it is called.
• This works as follows:
1) Initially, all routines are kept on disk in a relocatable-load format.
2) Firstly, the main-program is loaded into memory and is executed.
3) When a main-program calls the routine, the main-program first checks to see whether the routine has been loaded.
4) If routine has been not yet loaded, the loader is called to load desired routine into memory.
5) Finally, control is passed to the newly loaded-routine.
• Advantages:
1) An unused routine is never loaded.
2) Useful when large amounts of code are needed to handle infrequently occurring cases.
3) Although the total program-size may be large, the portion that is used (and hence loaded) may be much smaller.
4) Does not require special support from the OS.
-It is the responsibility of the users to design their programs to take advantage of such a method.
-Operating systems may help the programmer, however, by providing library routines to implement dynamic loading.

4.1.5 Dynamic Linking and Shared Libraries

• Static linking,
• system language libraries are treated like any other object module and are combined by the loader into the binary program image.
• This requirement wastes both disk space and main memory.

• Dynamic linking:
The concept of dynamic linking is similar to that of dynamic loading.
• Linking postponed until execution-time.
• A stub is included in the image for each library-routine reference.
• The stub is a small piece of code used to locate the appropriate memory-resident library-routine.
• When the stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory.
• Stub replaces itself with the address of the routine, and executes the routine.
• Thus, the next time that particular code-segment is reached, the library-routine is executed directly, incurring no cost for dynamic-linking.
• All processes that use a language library execute only one copy of the library code.

• Shared libraries:
• A library may be replaced by a new version, and all programs that reference the library will automatically use the new one.
• programs linked before the new library was installed will continue using the older library.
• This system is also known as shared libraries.
• Version info. is included in both program & library so that programs won't accidentally execute incompatible versions.

• Dynamic linking generally requires help from the operating system.
• If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

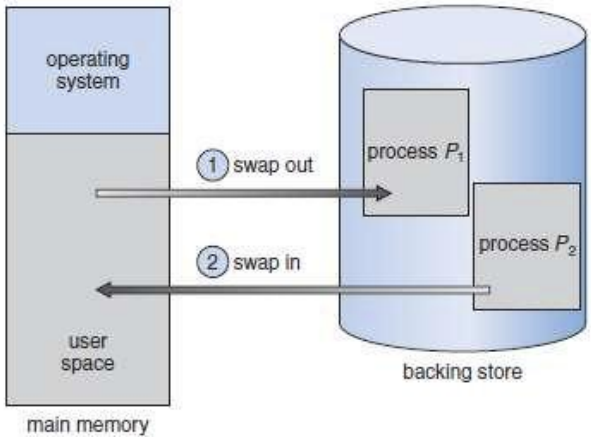
4.2 Swapping

• A process must be in memory to be executed.
• A process can be
→ swapped temporarily out-of-memory to a backing-store and
→ then brought into memory for continued execution.
• Backing-store is a fast disk which is large enough to accommodate copies of all memory-images for all users.

Why we need to swap out the process?

1) In Round RobinWhen
-a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed as shown in Figure.

2) Roll out/Roll in is a swapping variant used for priority-based scheduling algorithms.
-Lower-priority process is swapped out so that higher-priority process can be loaded and executed.
-Once the higher-priority process finishes, the lower-priority process can be swapped back in and continued



• Swapping depends upon address-binding:
1) If binding is done at load-time, then process cannot be easily moved to a different location.
2) If binding is done at execution-time, then a process can be swapped into a different memory space, because the physical-addresses are computed during execution-time.
• Major part of swap-time is transfer-time; i.e. total transfer-time is directly proportional to the amount of memory swapped.
• Disadvantages:
1) Context-switch time is fairly high.

2) If we want to swap a process, we must be sure that it is completely idle.
[reason:
-if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped.
-Assume that the I/O operation is queued because the device is busy.
- If we were to swap out process P1 and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P2.]

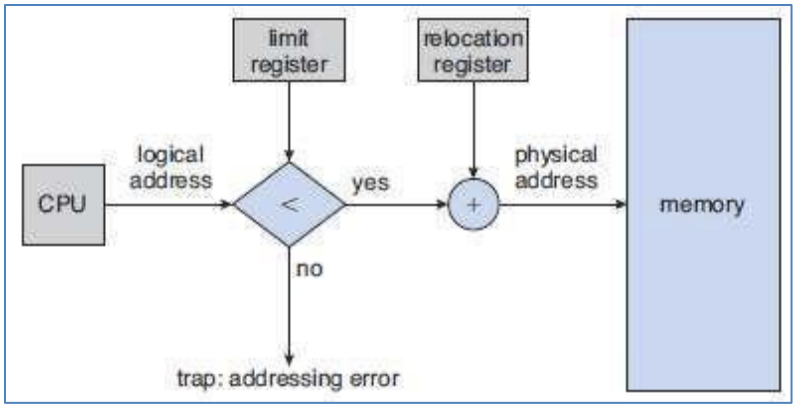
Two solutions:
i) Never swap a process with pending I/O.
ii) Execute I/O operations only into OS buffers.

4.3 Contiguous Memory Allocation

• Memory is usually divided into 2 partitions:
  → One for the resident OS.
  → One for the user-processes.
• Each process is contained in a single contiguous section of memory.

4.3.1 Memory Mapping & Protection

• Memory-protection means
  → protecting OS from user-process and
  → protecting user-processes from one another.
• Memory-protection is done using
  → Relocation-register: contains the value of the smallest physical-address.
  → Limit-register: contains the range of logical-addresses.
• Each logical-address must be less than the limit-register.
• The MMU maps the logical-address dynamically by adding the value in the relocation-register.
This mapped-address is sent to memory (as shown in Figure).



• When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit-registers with the correct values.
• Because every address generated by the CPU is checked against these registers, we can protect the OS from the running user process.
• The relocation-register scheme provides an effective way to allow the OS size to change dynamically.

• For example,
  -the operating system contains code and buffer space for device drivers.
  -If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes.
  -Such code is sometimes called transient operating-system code; it comes and goes as needed.
  -Thus, using this code changes the size of the operating system during program execution.

4.3.2 Memory Allocation

• One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
-The degree of multiprogramming is bound by the number of partitions.

• When a partition is free, a process is
→ selected from the input queue and
→ loaded into the free partition.

• When the process terminates, the partition becomes available for another process.

• The OS keeps a table indicating
→ which parts of memory are available and
→ which parts are occupied.

• A hole is a block of available memory.

• Normally, memory contains a set of holes of various sizes.

• Initially, all memory is
→ available for user-processes and
→ considered one large hole.

• In general, at any given time we have a *set* of holes of various sizes scattered throughout memory.

• When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.

• If the hole is too large, it is split into two parts.
- One part is allocated to the arriving process;
- The other is returned to the set of holes.

• When a process terminates, it releases its block of memory, which is then placed back in the set of holes.

• If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

• At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes.

• This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size *n* from a list of free holes.

• Three strategies used to select a free hole from the set of available holes.

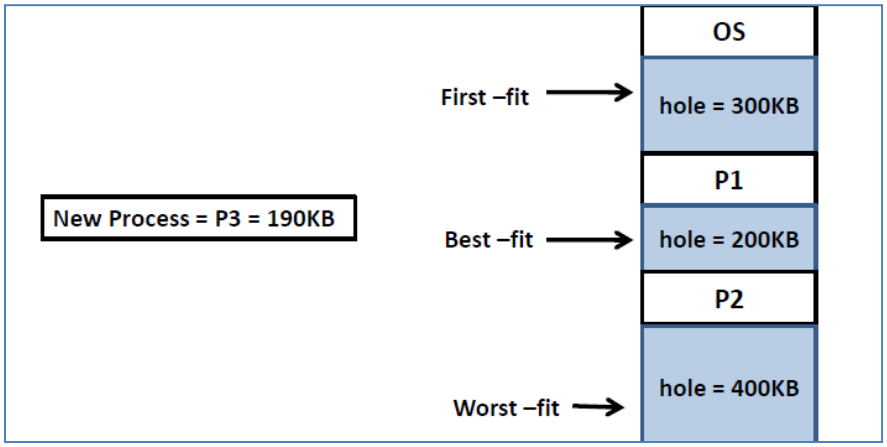
1) First Fit
-Allocate the first hole that is big enough.
- Searching can start either
→ at the beginning of the set of holes or
→ at the location where the previous first-fit search ended.

2) Best Fit
- Allocate the smallest hole that is big enough.
-We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

3) Worst Fit
- Allocate the largest hole.
- Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole.

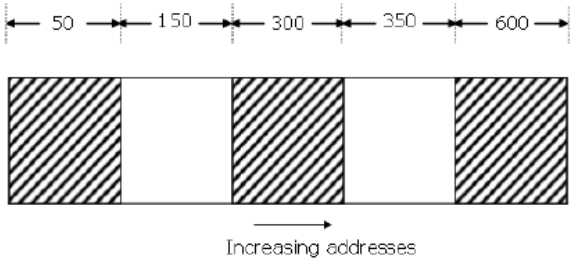
• First-fit and best fit are better than worst fit in terms of decreasing time and storage utilization.

Example 1:

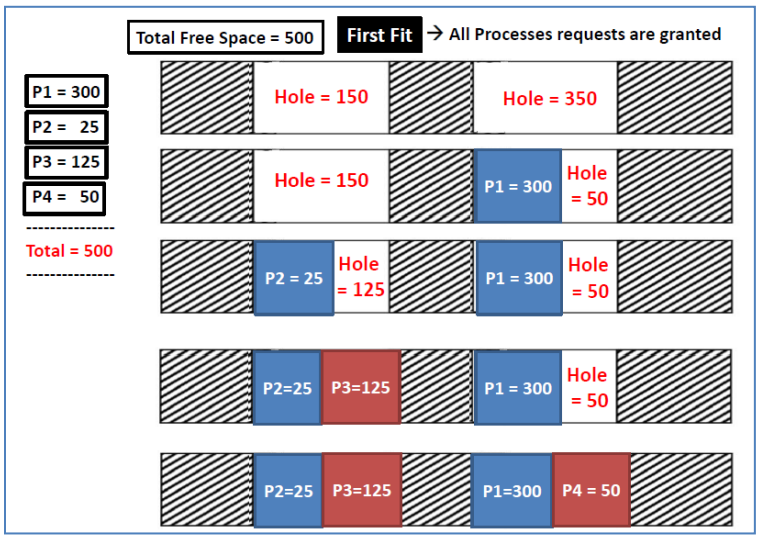


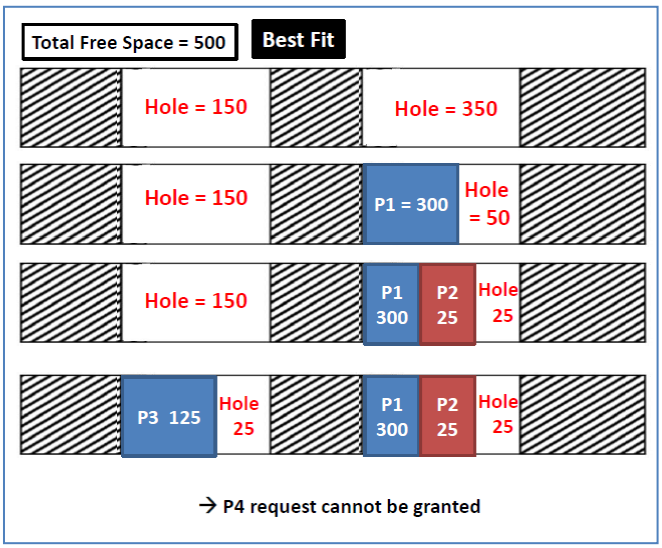
Example 2:

• Consider the following heap (figure) in which blank regions are not in use and hatched region are in use.



• The sequence of requests for blocks of size 300, 25, 125, 50 .





Worst fit :Same allocation will happen as with first fit.

4.3.3 Fragmentation

• Two types of memory fragmentation: 1) Internal fragmentation and
2) External fragmentation

1) Internal Fragmentation

• The general approach is to
→ break the physical-memory into fixed-sized blocks and
→ allocate memory in units based on block size

• The allocated-memory to a process may be slightly larger than the requested-memory.

• The difference between requested-memory and allocated-memory is called internal fragmentation i.e. Unused memory that is internal to a partition.

2) External Fragmentation

• External fragmentation occurs when there is enough total memory-space to satisfy a request but the available-spaces are not contiguous. (i.e. storage is fragmented into a large number of small holes).

• Both the first-fit and best-fit strategies for memory-allocation suffer from external fragmentation.

• Statistical analysis of first-fit reveals that
- given N allocated blocks, another 0.5 N blocks will be lost to fragmentation.
-This property is known as the 50-percent rule.

• Two solutions to external fragmentation :

1) Compaction

-The goal is to shuffle the memory-contents to place all free memory together in one large hole.

Compaction is possible only if relocation is
→ dynamic and
→ done at execution-time.

2) Permit the logical-address space of the processes to be non-contiguous.

- This allows a process to be allocated physical-memory wherever such memory is available.

-Two techniques achieve this solution:
1) Paging and
2) Segmentation.

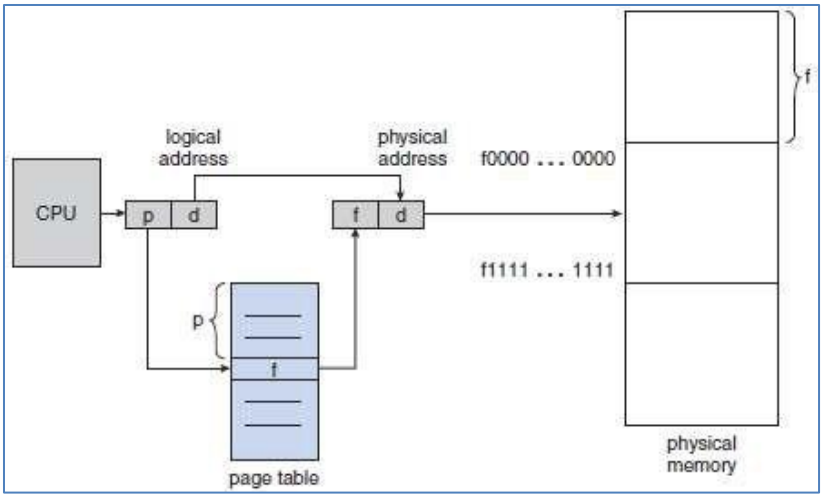
4.4. Paging

• Paging is a memory-management scheme.
• This permits the physical-address space of a process to be non-contiguous.
• This also solves the considerable problem of fitting memory-chunks of varying sizes onto the backing-store.
• Traditionally: Support for paging has been handled by hardware.
• Recent designs: The hardware & OS are closely integrated.

4.4.1. Basic Method

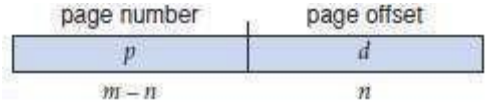
• Physical-memory is broken into fixed-sized blocks called frames.
• Logical-memory is broken into same-sized blocks called pages.
• The backing-store is divided into fixed-sized blocks that are of the same size as the memory-frames.
• When a process is to be executed, its pages are loaded into any available memory-frames from the backing-store.

• The hardware support for paging is illustrated in Figure
• Every address generated by the CPU is divided into two parts:
- a page number (p) and
- a page offset (d).
• The page number is used as an index into a page table.
• The page table contains the frame number of each page in physical memory
• Then, base address= frame no. x page size • This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.



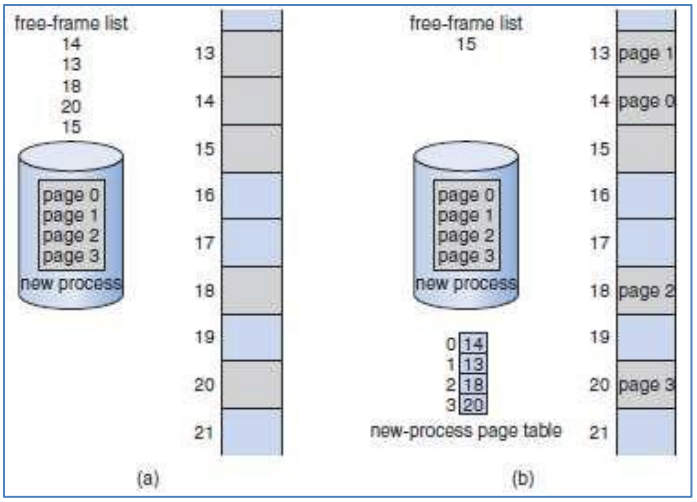
• The page size (like the frame size) is defined by the hardware.
• The size of a page is typically a power of 2, varying between 512 bytes and 16 MB per page, depending on the computer architecture.
• The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy.
• If -the size of logical address space is 2^m and
-a page size is 2^n addressing units (bytes or words), then
-the high-order $m - n$ bits of a logical address designate the page number, and
- the n low-order bits designate the page offset.

Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page.

• When a process arrives in the system to be executed, its size, expressed in pages, is examined.
• Each page of the process needs one frame.
• Thus, if the process requires *n* pages, at least *n* frames must be available in memory.
• If *n* frames are available, they are allocated to this arriving process.
• The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process.
• The next page is loaded into another frame, and its frame number is put into the page table, and so on as shown in Figure below .
• Free frames (a) before allocation and (b) after allocation



➢ External fragmentation and internal fragmentation:

• With paging scheme: no external fragmentation
• Any free frame can be allocated to a process that needs it.
• may have some internal fragmentation.
• Notice that frames are allocated as units.
• If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full.
• For example,

If page size is 2,048 bytes, a process of 72,766 bytes would need 35 pages plus 1,086 bytes.
It would be allocated 36 frames, resulting in an internal fragmentation of 2,048 - 1,086 = 962 bytes.

• In the worst case, a process would need *n* pages plus 1 byte. It would be allocated, *n* + 1 frames, resulting in an internal fragmentation of almost an entire frame.

➢ Paging example(as shown in figure below)

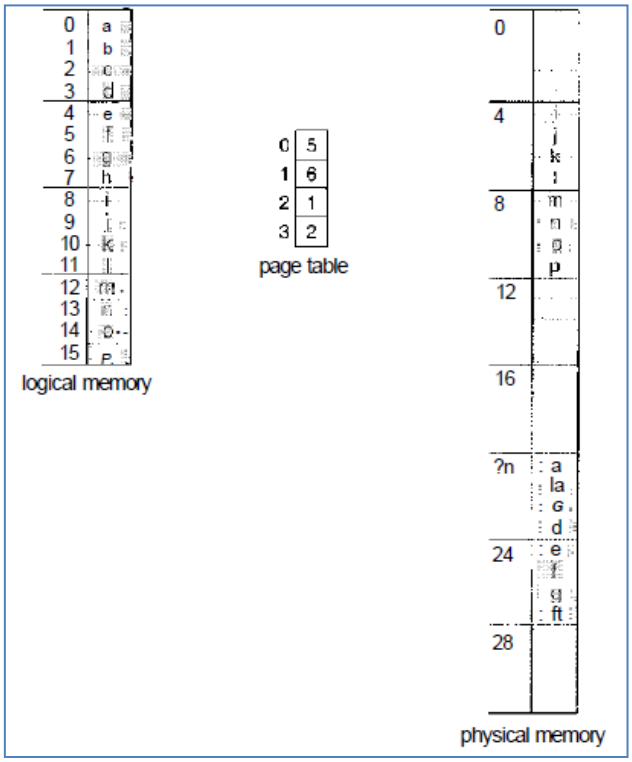
• Page size 4bytes

-2^2 bytes, hence 2 bits of logical address represent page no

• a 32-byte memory (32bytes/4bytes=8 pages)
• process has 4 pages

-hence 4*4 =16 bytes, so 16 logical address i.e, 0 to 15,
-hence 4 bit logical address(2 bit page no. and 2 bit offset that has to be added to base address in the pagetable)

• hence page table has 4 entries (2bit sufficient to represent page number i.e 0,1,2,3)
• page table contains the base address of each page in physical memory
• this base address is add with the offset in the logical address to get the exact location in physical location.



i.e logical address = 3 (address of 'd')

-in binary = 0011

-1st two bits (00) is the page no i.e, page 0 and

-in page table for page0 , 5 is the frame no

- hence base address is i.e 5x4(page size)=20

-Next 2 bits (11) of logical address i.e, 3 the offset is added to base address 20 , i.e., 20+3=23 which is the is the physical address of letter 'd'

- i.e logical address 3 maps to (5x4)+3=23

• similarly, logical address 4 maps to (6x4)+0=24 (4 in binary 0100 , so page no. 1 and offset 00 ,page table entry is 6]

➢ Frame Table:

• Paging provides clear separation between the user's view of memory and the actual physical memory.
• The user program views memory as one single space, containing only this one program.
• In fact, the user program is scattered throughout physical memory, which also holds other programs.
• The logical addresses are translated into physical addresses.
• This mapping is hidden from the user and is controlled by the operating system.

• Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—

- which frames are allocated,
- which frames are available,
- how manv total frames there are, and so on.

• This information is generally kept in a data structure called a frame table.
• The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

• In addition, the operating system must be aware that user processes operate in user space, and all logical addresses must be mapped to produce physical addresses.
• The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents, so, that address must be mapped to produce the correct physical address.

4.4.2 Hardware Support for Paging

The hardware implementation of the page table can be done in several ways.

Solution 1:

• the page table is implemented as a set of dedicated registers.
• These registers should be built with very high-speed logic to make the paging-address translation efficient.
• Changing page tables requires changing all these registers, hence more context-switch time.
• The use of registers for the page table is satisfactory if the page table is reasonably small (for example, 256 entries).
• Normally computers, however, allow the page table to be very large (for example, 1 million entries).
• For these machines, the use of fast registers to implement the page table is not feasible.

Solution 2:

• The page table is kept in main memory, and a page-table base register (PTBR) points to the page table.

• Changing page tables requires changing only this one register, substantially reducing context-switch time.

Problem:

• The problem with this approach is the time required to access a user memory location.

• If we want to access location i, we must first index into the page table, using the value in the PTBR offset by the page number.This task requires a memory access.

• It provides us with the frame number, which is combined with the page offset to produce the actual address.

• Then this actual address is used to access the desired place in memory.

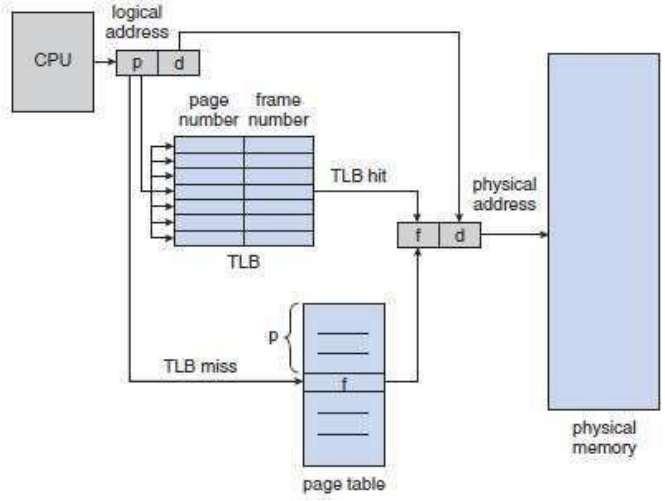
• With this scheme, *two* memory accesses are needed to access a byte (one for the page-table entry, one for the byte).

• Thus, memory access is slowed by a factor of 2.

This delay would be intolerable under most circumstances.

Solution 3:

• The standard solution to this problem is to use a special, small, fastlookup hardware cache, called a translation look-aside buffer (TLB).



TLB

• The TLB is associative, high-speed memory.

• Each entry in the TLB consists of two parts:

- a key (or tag) and
- a value.

• When the associative memory is presented with an item, the item is compared with all keys simultaneously.

• If the item is found, the corresponding value field is returned.

• The search is fast; the hardware, however, is expensive.

TLB with Paging:

The TLB is used with page tables(as shown in figure) in the following way:

• The TLB contains only a few of the page-table entries.

• When a logical address is generated by the CPU, its page number is presented to the TLB.

• If the page number is found (known as a TLB hit), its frame number is immediately available and is used to access memory.

• If the page number is not in the TLB (known as a TLB miss),

-a memory reference to the page table must be made.

-When the frame number is obtained, we can use it to access memory.

-In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

- If the TLB is already full of entries, the operating system must select one for replacement. Replacement policies range from least recently used (LRU) to random.

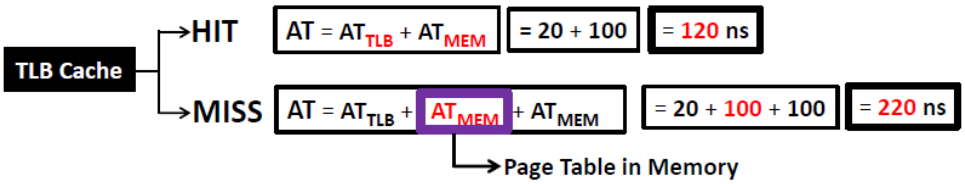
• Some TLBs allow entries to be wired down, meaning that they cannot be removed from the TLB.
• Typically, TLB entries for kernel code are wired down.

• Some TLBs store address-space identifiers (ASIDs) in each TLB entry.
• An ASID uniquely identifies each process and is used to provide address-space protection for that process.
• When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running process matches the ASID associated with the virtual page.
• If the ASIDs do not match, the attempt is treated as a TLB miss.
• an ASID allows the TLB to contain entries for several different processes simultaneously.
• If the TLB does not support separate ASIDs, then every time a new page table is selected (for instance, with each context switch), the TLB must be flushed (or erased) to ensure that the next executing process does not use the wrong translation information.

EFFECTIVE ACCESS TIME:

• The percentage of times that a particular page number is found in the TLB is called the hit ratio.
  -An 80-percent hit ratio means that we find the desired page number in the TLB 80 percent of the time.
• If it takes
  -search time in TLB: 20 nanoseconds
  - Memory access time: 100 nanoseconds to access memory,
  -then a mapped-memory access takes 120 nanoseconds when the page number is in the TLB.

• If we fail to find the page number in the TLB (20 nanoseconds), then we must first access memory for the page table and frame number (100 nanoseconds) and then access the desired byte in memory (100 nanoseconds), for a total of 220 nanoseconds.



• To find the effective memory-access time, we weight each case by its probability:
Example 1:
If 80% hit ratio: hence 20% miss ratio

effective access time = 0.80 x 120 + 0.20 x 220 = 140 nanoseconds.

In this example, we suffer a 40-percent slowdown in memory-access time (from 100 to 140 nanoseconds).

Example 2:
• For a 98-percent hit ratio,
  -effective access time = 0.98 x 120 + 0.02 x 220 = 122 nanoseconds.
• This increased hit rate produces only a 22 percent slowdown in access time.

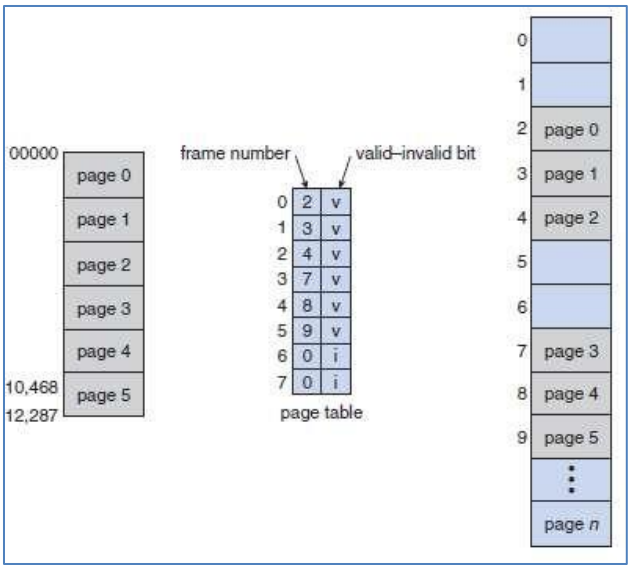
4.4.3 Protection

• Memory-protection is achieved by protection-bits for each frame.
• The protection-bits are kept in the page-table.
• One protection-bit can define a page to be read-write or read-only.
• Every reference to memory goes through the page-table to find the correct frame-number.
• Firstly, the physical-address is computed. At the same time, the protection-bit is checked to verify that no writes are being made to a read-only page.
• An attempt to write to a read-only page causes a hardware-trap to the OS (or memory-protection violation).

Valid Invalid Bit : to trap illegal addresses

• This bit is attached to each entry in the page-table as shown in the figure below
  1) Valid bit: The page is in the process' logical-address space.
  2) Invalid bit: The page is not in the process' logical-address space.
• The OS sets this bit for each page to allow or disallow access to the page.

Example:
Given:
• a system has a 14-bit address space (0 to 16383 i.e, $(2^{14} - 1)$),
• a program that use only addresses 0 to 10468 (i.e, 10469 bytes)
• a page size of 2 KB(2048 bytes),
• Then program requires 6 pages (10469/2048= 5.11 =6pages)
• thus we get the situation shown in Figure .
• Addresses in pages 0,1,2,3,4, and 5 are mapped normally through the page table and the valid-invalid bit is set to valid
• Any attempt to generate an address in pages 6 or 7, however, will find that the valid-invalid bit is set to invalid, and the computer will trap to the operating system (invalid page reference).



Disadvantage:
1)• Because the program extends to only address 10468, any reference beyond that address is illegal.
  • However, references to page 5 are classified as valid, so accesses to addresses up to 12287 are valid.
  • This problem is a result of the 2-KB page size and the internal fragmentation of paging.
2) • Many processes use only a small fraction of the address space available to them.
  • It would be wasteful in these cases to create a page table with entries for every page in the address range. As these would take up valuable memory space.

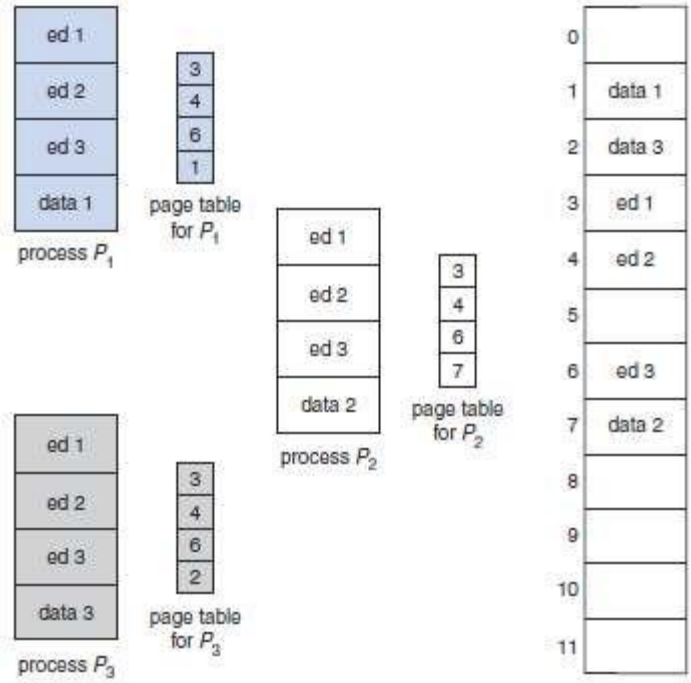
Solution:
• a page-table length register (PTLR), to indicate the size of the page table.
• This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

4.4.4 Shared Pages

• An advantage of paging is the possibility of *sharing* common code.
• This consideration is particularly important in a time-sharing environment.

• Consider a system that supports 40 users, each of whom executes a text editor.
If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB (200KBx40) to support the 40 users.

• If the code is reentrant code (or pure code) (Reentrant code is non-self-modifying code; it never changes during execution.) , however, it can be shared, as shown in Figure below.
• Here we see a three-page editor—each page 50 KB in size shared among three processes.
• Each process has its own data page.
• Since the editor code is reentrant code , two or more processes can execute the same code at the same time.
• Each process has its own copy of registers and data storage to hold the data for the process's execution.
• Only one copy of the editor need be kept in physical memory.
• Each user's page table maps onto the same physical copy of the editor, but data pages are mapped onto different frames.
• Thus, to support 40 users, we need only one copy of the editor (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB—a significant savings.



• Other heavily used programs can also be shared—compilers, window systems, run-time libraries, database systems, and so on.

4.5 Structure of the Page Table

1) Hierarchical Paging
2) Hashed Page-tables
3) Inverted Page-tables

4.5.1 Hierarchical Paging

• Problem: Most computers support a large logical-address space (2^{32} to 2^{64}).
• In these systems, the page-table itself becomes excessively large.
• Solution: Divide the page-table into smaller pieces.

➢ Two Level Paging Algorithm

Consider a system with a 32-bit logical address space.

-total logical memory reduired: 2^{32} bytes (4 GB)

- page size: 4 KB (2^{12} bytes),

- number of pages : $2^{32}/2^{12} = 2^{20}$ (~1 million pages) (20 bits sufficient to represent a page number)

- thus 1 million entries in the page table

- thus in 32 bit logical address,

-20 bit represent page number and

-12 bit page offset

-size of page table entry(PTE): 4 bytes

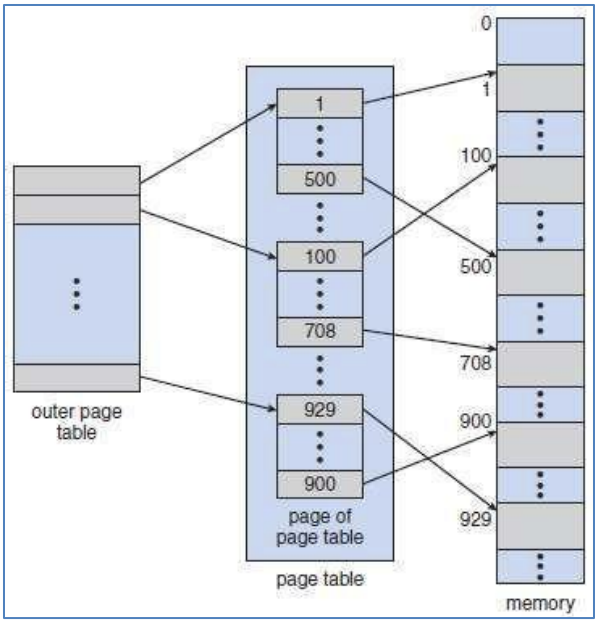
-thus page table size = 4* 2^{20} bytes = 4 MB

-thus a process may need up to 4 MB of physical address space for the page table alone

-It is not possible to allocate the page table contiguously in main memory.

• Solution: a two-level paging algorithm,

- the page table itself is also paged as shown in figure



-A logical address is divided into

-a page number consisting of 20 bits and

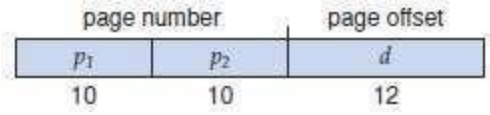
-a page offset consisting of 12 bits.

Since the page table itself paged, the page number is further divided into

-a 10-bit page number and

-a 10-bit page offset.

Thus, a logical address is as follows:

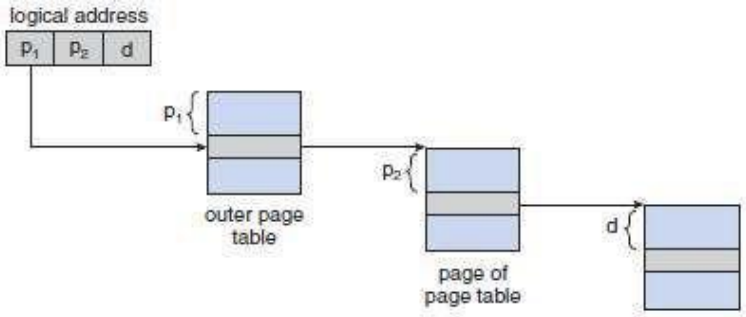


where

-*p1* is an index into the outer page table

-*p2* is the displacement within the page of the outer page table.

• The address-translation method for this architecture is shown in Figure below.

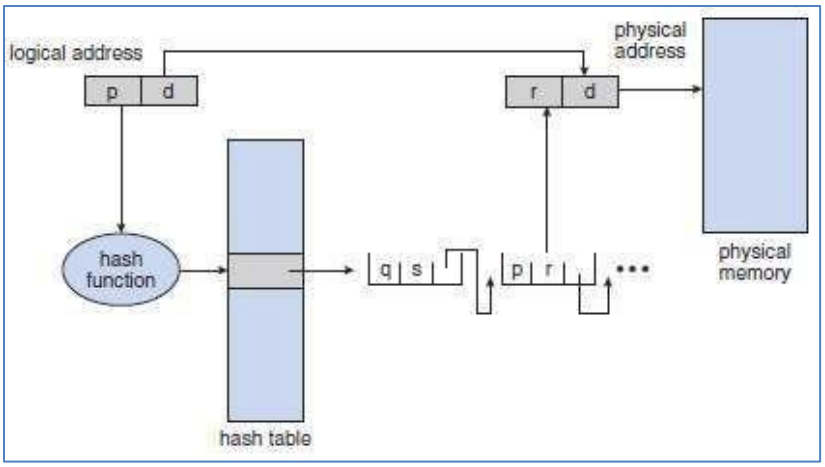


• Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.

4.5.2 Hashed Page Tables

• used for handling address spaces larger than 32 bits.
• The hash-value is the virtual page-number.
• Each entry in the hash-table contains a linked-list of elements that hash to the same location (to handle collisions).
• Each element consists of 3 fields:
1) Virtual page-number
2) Value of the mapped page-frame and
3) Pointer to the next element in the linked-list.
• The algorithm works as follows a shown in figure :

1) The virtual page-number in the virtual address is hashed into the hash-table.
2) The hash value, i.e, virtual page-number is compared with the first element in the linked-list.
3) If there is a match, the corresponding page-frame (field 2) is used to form the desired physical-address.
4) If there is no match, subsequent entries in the linked-list are searched for a matching virtual page-number.



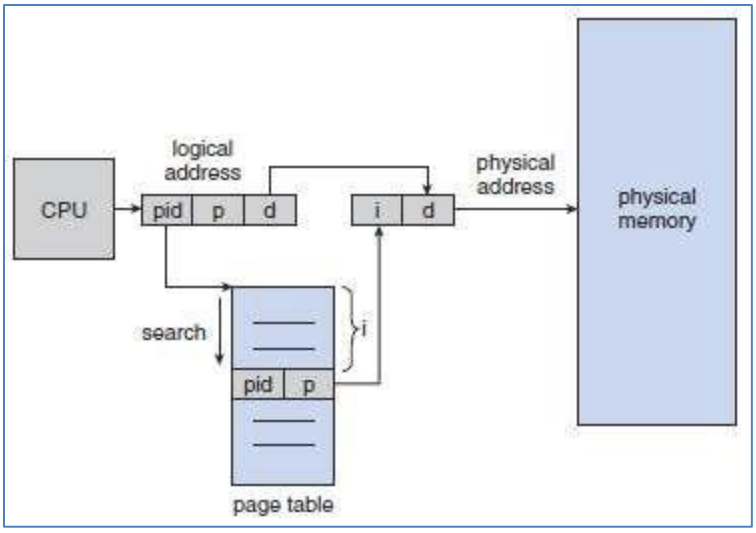
➢ Clustered Page Tables

• Favorable for 64-bit address spaces
• These are similar to hashed page-tables except that each entry in the hash-table refers to several pages rather than a single page.
• Advantages:
- Useful for address spaces, where memory-references are noncontiguous and scattered throughout the address space.

4.5.3 Inverted Page Tables

-each process has an associated page table.
- page tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.
Solution : Inverted Page Tables
• Has one entry for each real page of memory.
• Each entry consists of
→ virtual-address of the page stored in that real memory-location and
→ information about the process that owns the page.
-Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame
• Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

Example: inverted page table used in the IBM RT as shown in figure:



• Each virtual-address consists of a triplet :
<process-id, page-number, offset>.

• Each inverted page-table entry is a pair
<process-id, page-number>

• The algorithm works as follows:
1) When a memory-reference occurs, part of the virtual-address, consisting of <process-id, page-number>, is presented to the memory subsystem.
2) The inverted page-table is then searched for a match.
3) If a match is found, at entry 'i' -then the physical-address <i, offset> is generated.
4) If no match is found, then an illegal address access has been attempted.

• Advantage:
1) Decreases memory needed to store each page-table

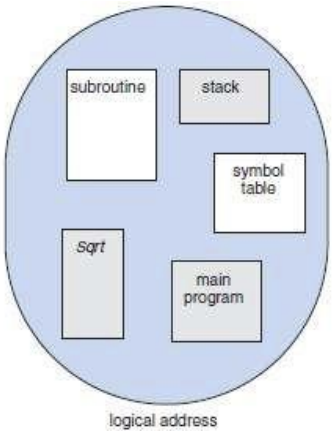
• Disadvantages:
1) Increases amount of time needed to search table when a page reference occurs.
2) Difficulty implementing shared-memory.
-because there is only one virtual page entry for every physical page,
-one physical page cannot have two (or more) shared virtual addresses.

4.6 Segmentation

• A user while writing the program thinks of it as a main program with a set of methods, procedures, or functions.
• It may also include various data structures: objects, arrays, stacks, variables, and so on.
• Each of these modules or data elements is referred to by name.
• Each of these segments is of variable length;

4.6.1 Basic Method

• Segmentation is a memory-management scheme that supports above discussed user-view of memory as shown in figure
• A logical-address space is a collection of segments.
• Each segment has a name and a length.
• The addresses specify both
→ segment-name and
→ offset within the segment.



For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.
Thus, a logical address consists of a *two tuple:*
< segment-number, offset >.

• Normally, the user-program is compiled, and the compiler automatically constructs segments reflecting the input program.

For ex A C compiler might create separate segments for the following :

→ The code
→ The heap, from which memory is allocated
→ The standard C library
→ Global variables
→ The stacks used by each thread

4.6.2 Hardware Support

• Segment-table maps 2 dimensional user-defined addresses into one-dimensional physical-addresses.
• In the segment-table, each entry has following 2 fields:

1) Segment-base : contains starting physical-address where the segment resides in memory.
2) Segment-limit : specifies the length of the segment

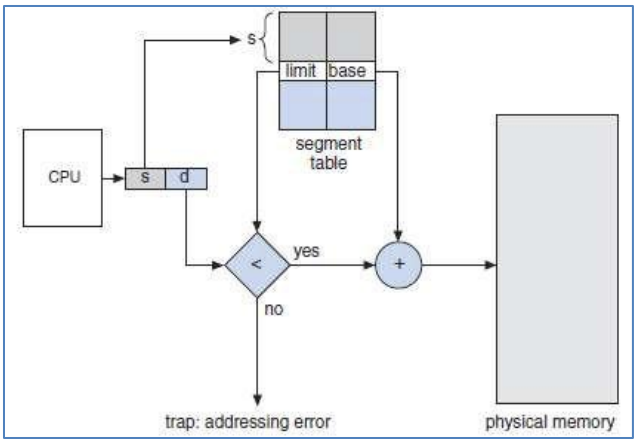
➢ The use of a segment table is illustrated in Figure:

• A logical-address consists of 2 parts:

1) Segment-number(s) :
-used as an index to the segment-table .
2) Offset(d) of the logical address
-must be between 0 and the segment-limit.

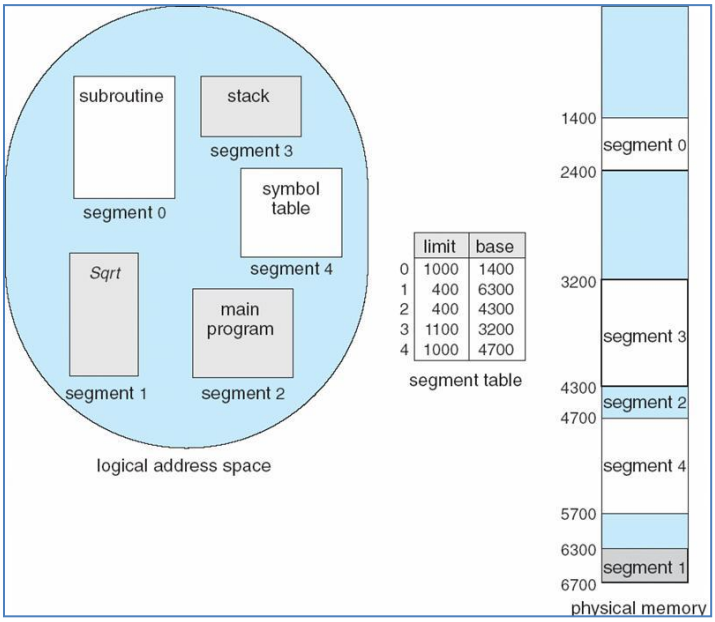
• If offset is not between 0 & segment-limit, then we trap to the OS(logical-addressing attempt beyond end of segment).
• If offset is legal, then it is added to the segment-base to produce the physical-memory address.

The segment table is thus essentially an array of base-limit register pairs.



➢ Example of segmentation:

• As an example, consider the situation shown in Figure .
-there are five segments numbered from 0 through 4.
-The segments are stored in physical memory as shown in figure.
-The segment table has a separate entry for each segment, giving
-the beginning address of the segment in physical memory (or base) and
-the length of that segment (or limit).
• For example,
• segment 2 : Limit = 400 bytes, Base = 4300.
-Thus, a reference to byte 53 which is less than limit(400) is mapped on to location :
base+ 53 = 4300 + 53 = 4353.



• similarly a reference to segment 3's byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052.

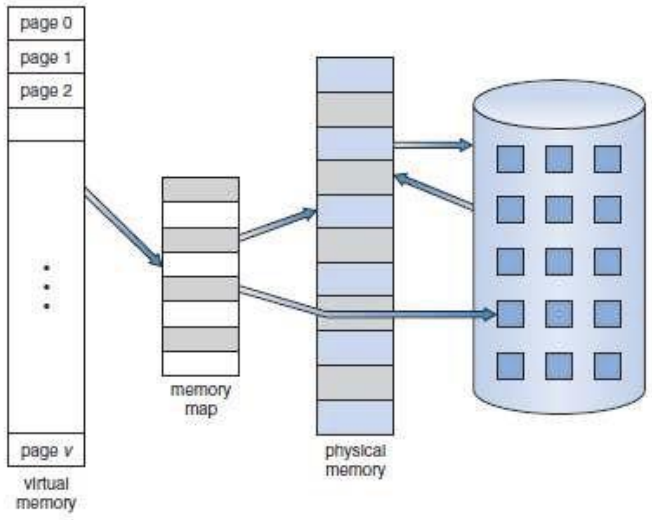
• A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

VIRTUAL MEMORY

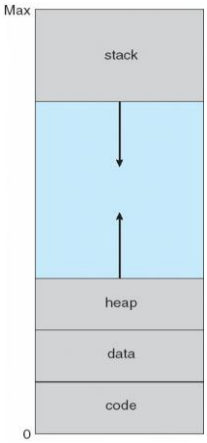
4.7 Virtual Memory: Background

• The instructions being executed must be in physical memory.
• but it is also unfortunate, since it limits the size of a program to the size of physical memory.
• In many cases, the entire program is not needed.
• For example:
→ Unusual error-conditions are almost never executed.
→ Arrays,lists and tables are often allocated more memory than needed.
→ Certain options & features of a program may be used rarely.
• Even in those cases where the entire program is needed, it may not all be needed at the same time.
• The ability to execute a program that is only partially in memory would confer many benefits:
- More programs could be run at the same time.
- Programmers could write for a large virtual-address space and need no longer use overlays.
- Less I/O would be needed to load/swap programs into memory, so each user program would run faster.

• Virtual Memory is a technique that allows the execution of processes that are not completely in memory as shown in Figure.

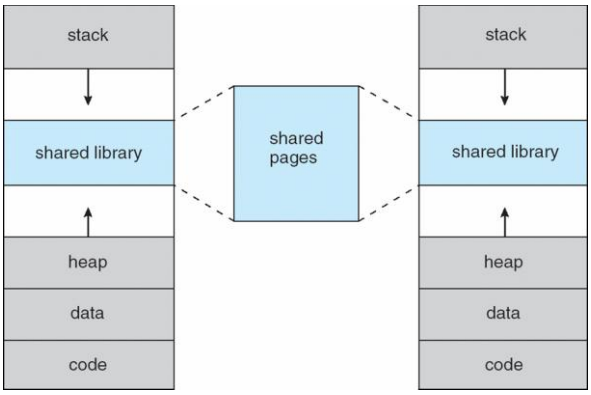


• VM involves the separation of logical-memory as perceived by users from physical-memory.
- The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory.
-Virtual address spaces that include holes are known as sparse address spaces.
- Example: a hole between stack and heap as shown in figure… require actual physical pages only if the heap or stack grows.



• VM allows files and memory to be shared by several processes through page-sharing thus following benefits:

- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
-virtual memory enables processes to share memory.
- Virtual memory allows one process to create a region of memory that it can share with another process.
- Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Figure.



• Virtual memory can allow pages to be shared during process creation with the fork ( ) system call, thus speeding up process creation.

• It is up to the MMU to map logical-pages to physical page-frames in memory.

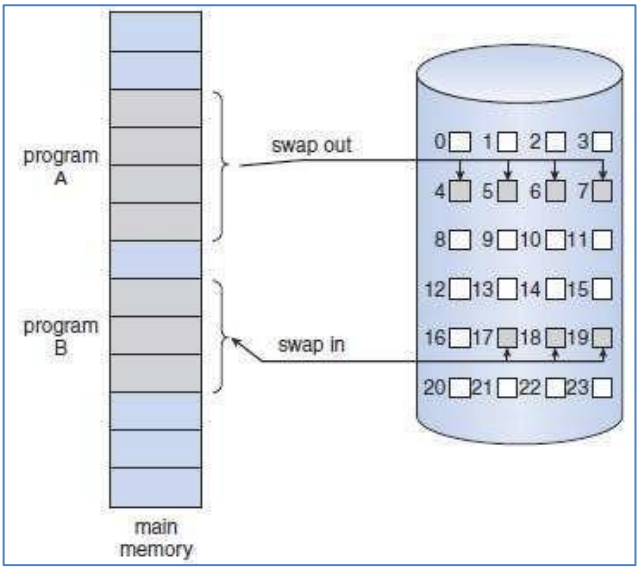
• Virtual-memory can be implemented by:

1) Demand paging and
2) Demand segmentation.

4.8 Demand Paging

• With demand-paged virtual memory, pages are only loaded when they are demanded during program execution;
• pages that are never accessed are thus never loaded into physical memory.

• A demand-paging system is similar to a paging-system with swapping (as shown in Figure)
• Processes reside in secondary-memory (usually a disk).
• When we want to execute a process, we swap it into memory.
• Instead of swapping in a whole process, lazy swapper brings only those necessary pages into memory.



• Since the lazy swapper swaps a page , its called pager with respect to demand paging.

4.8.1 Basic Concepts

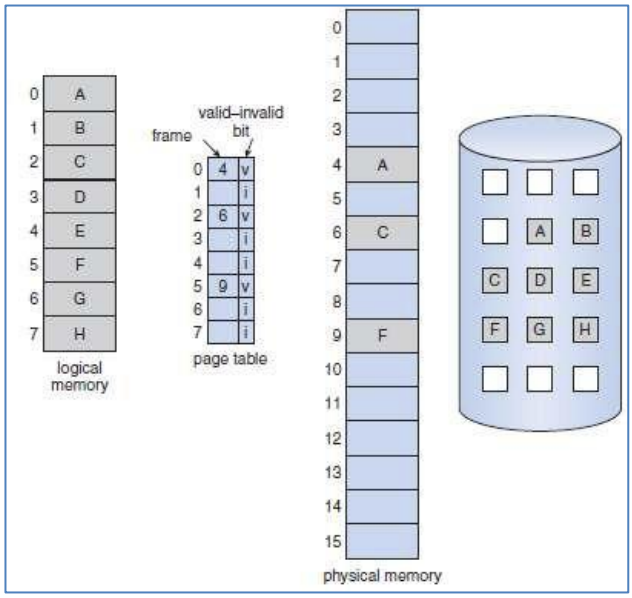
• Instead of swapping in a whole process, the pager brings only those necessary pages into memory.

•thus:

1) Avoids reading into memory-pages that will not be used,

2) Decreases the swap-time and

3) Decreases the amount of physical-memory needed.



• The valid-invalid bit scheme can be used to distinguish between → pages that are in memory and

→ pages that are on the disk.

1) If the bit is set to valid, the associated page is both legal and in memory.

2) If the bit is set to invalid, the page either

→ is not valid (i.e. not in the logical-address space of the process) or

→ is valid but is currently on the disk

• This is illustrated in the Figure.

• Marking a page invalid will have no effect if the process never attempts to access that page.

• While the process executes and accesses pages that are memory resident, execution proceeds normally.

• Access to a page marked invalid causes a page-fault trap.

Thus:

• A page-fault occurs when the process tries to access a page that was not brought into memory.

• Procedure for handling the page-fault (as shown in figure):

1) Check an internal-table (kept in PCB) to determine whether the reference was a valid or an invalid memory access.

2) If the reference is invalid, we terminate the process.

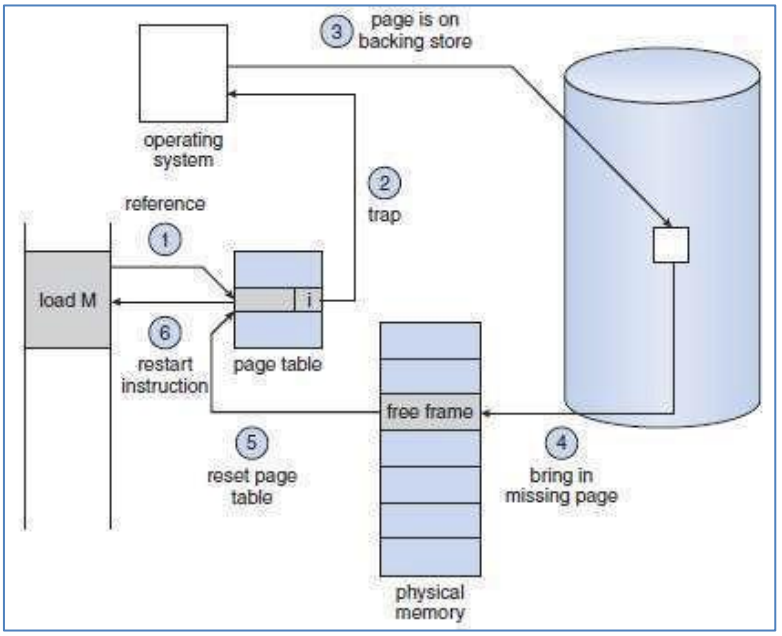
-If reference is valid, but we have not yet brought in that page, we now page it in.

3) Find a free-frame (by taking one from the free-frame list, for example).

4) Read the desired page into the newly allocated frame.

5) Modify the internal-table and the page-table to indicate that the page is now in memory.

6) Restart the instruction that was interrupted by the trap.



• **Pure demand paging**: Never bring pages into memory until required.
• Some programs may access several new pages of memory with each instruction, causing multiple page-faults and poor performance.
• Programs tend to have locality of reference, so this results in reasonable performance from demand paging.
• Hardware support:

1) Page-table
-Mark an entry invalid through a valid-invalid bit.
2) Secondary memory
- It holds pages that are not present in main-memory.
- It is usually a high-speed disk.
- It is known as the swap device and the section of disk used for this purpose is known as swap space.

• A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault.
• This is by no means the only architectural problem resulting from adding paging to an existing architecture to allow demand paging
• Paging is added between the CPU and the memory in a computer system.
• It should be entirely transparent to the user process.

4.8.2 Performance of Demand Paging

• Demand paging can significantly affect the performance of a computer-system.
• Let p be the probability of a page-fault ($0 \leq p \leq 1$).

- If p = 0, no page-faults.
- If p = 1, every reference is a fault.

• **Effective access time(EAT)=[(1 - p) *memory access]+ [p *page-fault time]**
• To compute the effective access time, we must know how much time is needed to service a page fault.
• A page-fault causes the following events to occur :

1) Trap to the OS.
2) Save the user-registers and process-state.
3) Determine that the interrupt was a page-fault. '
4) Check that the page-reference was legal and determine the location of the page on the disk.
5) Issue a read from the disk to a free frame:
   a. Wait in a queue for this device until the read request is serviced.
   b. Wait for the device seek time.
   c. Begin the transfer of the page to a free frame.
6) While waiting, allocate the CPU to some other user.
7) Receive an interrupt from the disk I/O subsystem (I/O completed).
8) Save the registers and process-state for the other user (if step 6 is executed).
9) Determine that the interrupt was from the disk.
10) Correct the page-table and other tables to show that the desired page is now in memory.
11) Wait for the CPU to be allocated to this process again.
12) Restore the user-registers, process-state, and new page-table, and then resume the interrupted instruction.

• Not all of these steps are necessary in every case.
• In any case, we are faced with three major components of the page-fault service time:
    1. Service the page-fault interrupt.
    2. Read in the page.
    3. Restart the process.
• The first and third tasks can be reduced, with careful coding, to several hundred instructions.
• Consider:
    - average page-fault service time : 8 milliseconds
    - memory-access time : 200 nanoseconds,
    - then the effective access time in nanoseconds is

$$\text{effective access time} = (1 - p) \times (200) + p \text{ (8 milliseconds)}$$
$$= (1 - p) \times 200 + p \times 8{,}000{,}000$$
$$= 200 + 7{,}999{,}800 \times p$$

• We see, that the effective access time is directly proportional to the page-fault rate.

Problem:

Let the page fault service time be 10ms in a computer with average memory access time being 20ns. If one page fault is generated for every 10^6 memory accesses, what is the effective access time for the memory?

GIVEN INPUT

Page-Fault service time= s = 10 **milliseconds** = 10×10^6 nanoseconds
Memory access time = ma =20 nanoseconds
Page Fault Rate = p = $1/10^6$
Effective Access Time = ?
EAT = (1 – p) x ma+ p x s
EAT = $(1 - 1/10^6) \times 20 + (1/10^6) \times 10^7$ nanoseconds
= (1 - 0.000001) x 20 + 10 nanoseconds
= (0.999999) x 20 + 10 nanoseconds
= 19.99998+ 10 nanoseconds
= 29.99998 nanoseconds

4.9 Copy-on-Write

• The fork() system call creates a child process as a duplicate of its parent.
• Traditionally, fork( ) worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
• However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary.
• Thus Copy-on-Write, technique allows the parent and child processes initially to share the same pages.
• If either process writes to a shared-page, a copy of the shared-page is created.

• **For example:**

- Assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
- OS will then create a copy of this page, mapping it to the address space of the child process.
- Child process will then modify its copied page & not the page belonging to the parent process.

• Copy-on-write is illustrated in Figures (a) and Figure (b), which show the contents of the physical memory before and after process 1 modifies page C.

Figures (a)

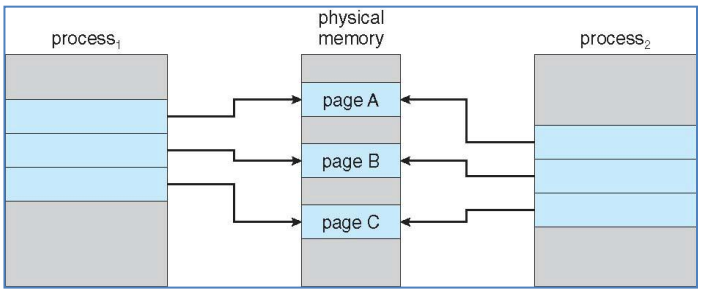
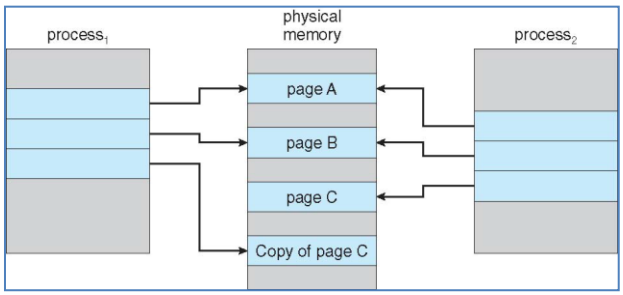


Figure (b)



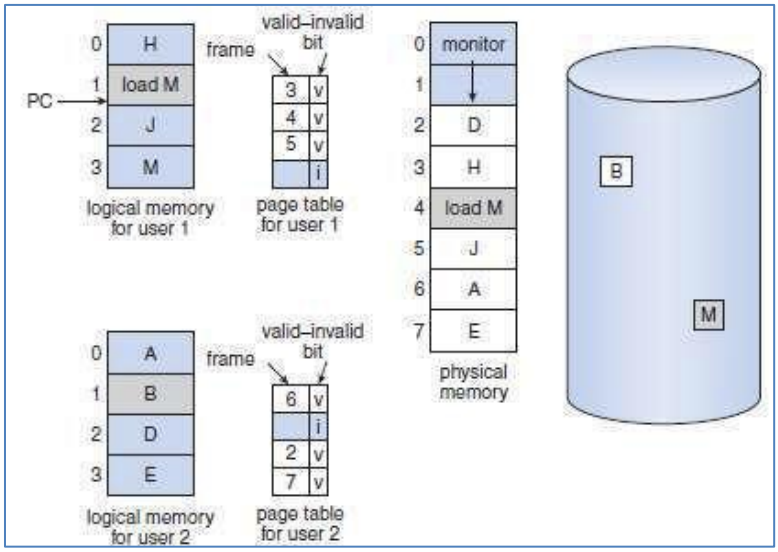
4.10 Page Replacement

• If we increase our degree of multiprogramming, we are over-allocating memory.

• While a user-process is executing, a page-fault occurs.

• The OS determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list

• This situation is illustrated in figure below when a page fault occurs for page B, there is no free frames.



Solution: page replacement.

• The OS then could:

→ Terminate the user-process.

→ Swap out a process, freeing all its frames, and reducing the level of multiprogramming.

→ Perform page replacement.

4.10.1 Basic Page Replacement

• Basic page replacement approach:

→ If no frame is free, we find one that is not currently being used and free it

• Page replacement takes the following steps(as shown in figure):

1) Find the location of the desired page on the disk.

2) Find a free frame:

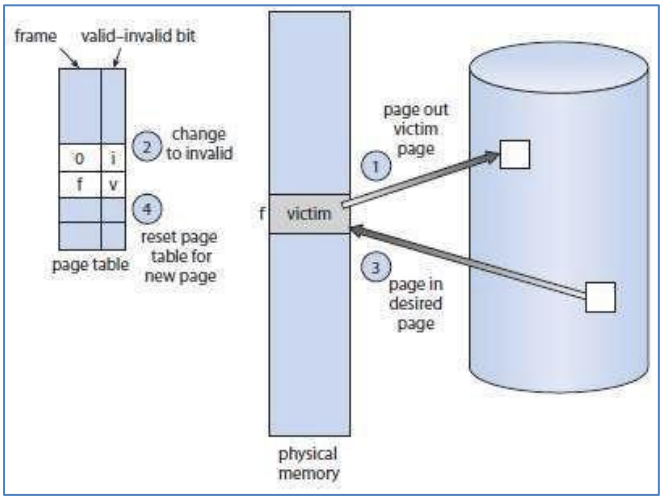
a. If there is a free frame, use it.

b. If there is no free frame, use a page-replacement algorithm to select a victim-frame.

c. Write the victim-frame to the disk; change the page and frame-tables accordingly.

3) Read the desired page into the newly freed frame; change the page and frame-tables.

4) Restart the user-process.



• If no frames are free, 2 page transfers (one out & one in) are required.

This situation
→ doubles the page-fault service-time and
→ increases the EAT accordingly.

Solution: Use a modify-bit (or dirty bit).

• Each page or frame has a modify-bit associated with the hardware.
• The modify-bit for a page is set by the hardware whenever any word is written into the page (indicating that the page has been modified).
• Working:

1) When we select a page for replacement, we examine its modify-bit.

2) If the modify-bit =1, the page has been modified. So, we must write the page to the disk.

3) If the modify-bit=0, the page has not been modified. So, we need not write the page to the disk, it is already there.

• Thus Can reduce the time required to service a page-fault.

• two major problems to be solved to implement demand paging:

1) Develop a Frame-allocation algorithm:

- to decide how many frames to allocate to each process if we have multiple processes in memory

2) Develop a Page-replacement algorithm:

- to select the frames that are to be replaced.

• There are many different page-replacement algorithms.
• we want the one with the lowest page-fault rate.
• To evaluate an algorithm, its runs on a particular string of memory references and computing the number of page faults.
• The string of memory references is called a **reference string.**

• **Note:**
• To reduce the number of data, we use two facts.
1) consider only the page number, rather than the entire address.
2) if we have a reference to a page *p*, then any *immediately* following references to page *p* will never cause a page fault since Page *p* will be in memory after the first reference

• **Converting addresses into page numbers:**
For example, if we trace a particular process, we might record the following address sequence:

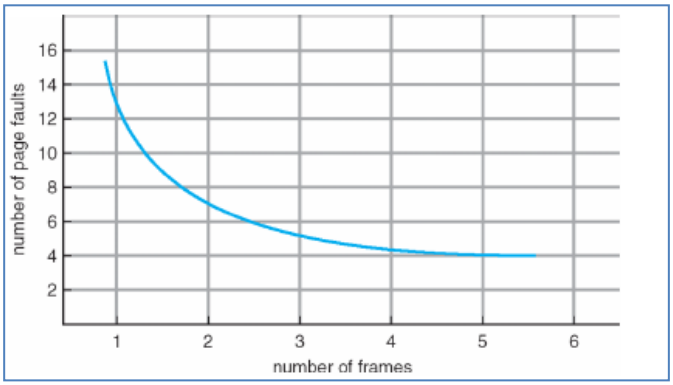
0100, 0432, 0101,0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104,0101,0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference string:

1,4,1,6,1,6,1,6,1,6,1

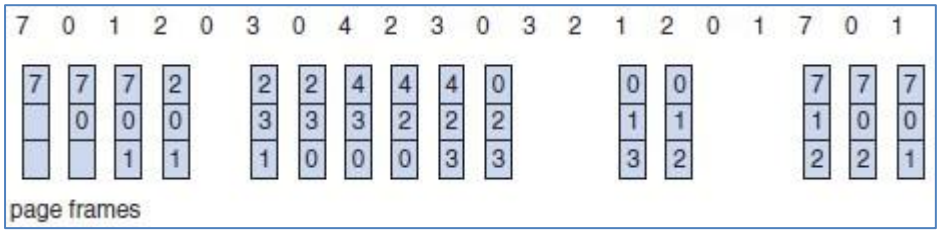
Address range	Page number
0 to 99	0
100 to 199	1
200 to 299	2
So on….	……

• To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the number of page frames available.
• As the number of frames available increases, the number of page faults decreases.
• In general, we expect a curve such as that in Figure .
• As the number of frames increases, the number of page faults drops to some minimal level.



4.10.2. FIFO Page Replacement

• Each page is associated with the time when that page was brought into memory.
• When a page must be replaced, the oldest page is chosen.
• We use a FIFO queue to hold all pages in memory as shown in figure.
When a page must be replaced, we replace the page at the head of the queue
When a page is brought into memory, we insert it at the tail of the queue.
• Example: Consider the following references string with frames initially empty.



-The first three references(7, 0, 1) cause page-faults and are brought into these empty frames.

- The next reference(2) replaces page 7, because page 7 was brought in first.
- Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line.
- This process continues till the end of string.
- There are fifteen faults altogether.

• Advantage:

1) Easy to understand & program.

• Disadvantages:

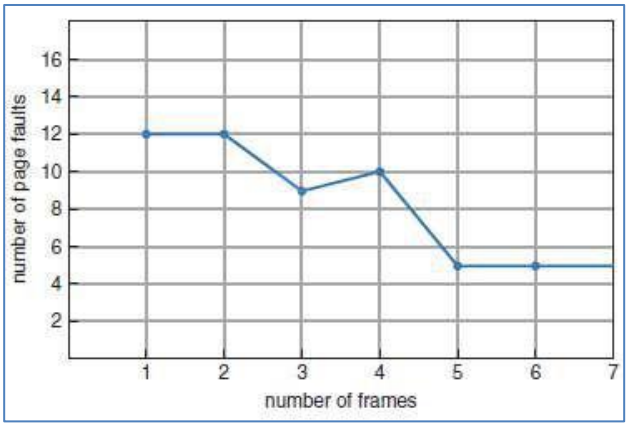
1) Performance is not always good

2) A bad replacement choice increases the page-fault rate (Belady's anomaly).

Belady's anomaly:

• For some algorithms, the page-fault rate may increase as the number of allocated frames increases. This is known as Belady's anomaly.

• Example: Consider the following reference string:
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



• Figure shows the curve of page faults for this reference string versus the number of available frames.

• We can see that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)

4.10.3 Optimal Page Replacement

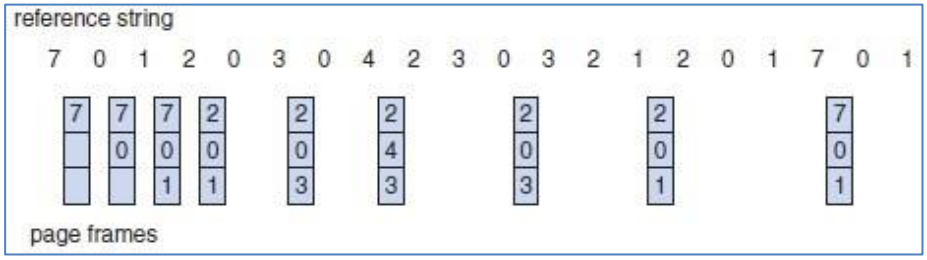
• Working principle:

Replace the page that will not be used for the longest period of time

• This is used mainly to solve the problem of Belady‟s Anamoly.

• This has the lowest page-fault rate of all algorithms.

• Consider the following reference string:

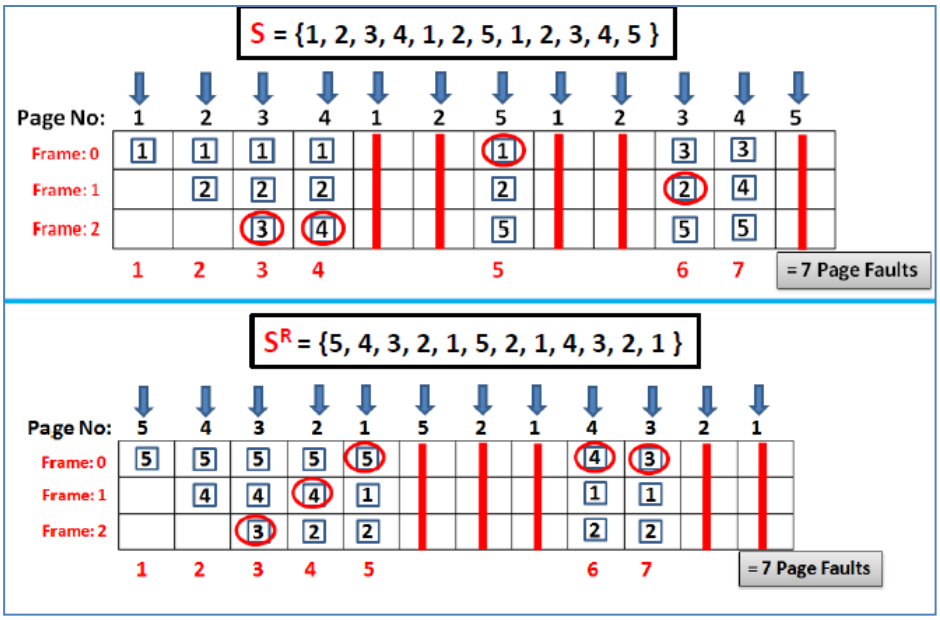


- The first three references cause faults that fill the three empty frames.
- The reference to page 2 replaces page 7, because
  -page 7 will not be used until reference 18.
  -the page 0 will be used at 5, and page 1 at 14.

-With only nine page-faults, optimal replacement is much better than a FIFO algorithm, which results in fifteen faults.

- **S - Reference String**
- **S^R - Reverse of Reference String**

➢ **No of Page Faults of "S" using OPTIMAL = No of Page Faults of "S^R" using OPTIMAL**

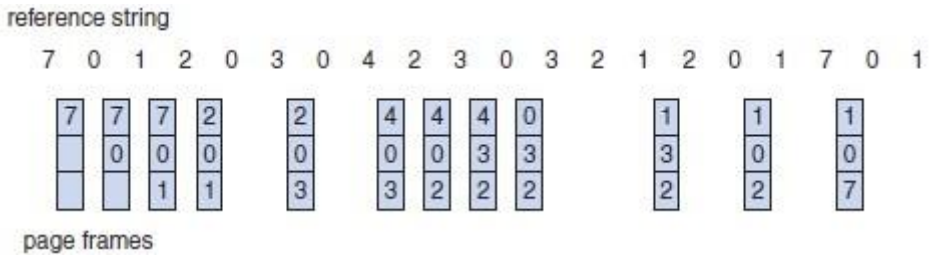


- Advantage:
  1) Guarantees the lowest possible page-fault rate for a fixed number of frames.
- Disadvantage:
  1) Difficult to implement, because it requires future knowledge of the reference string.

4.10.4 LRU (least-recently-used)age Replacement

- The key difference between FIFO and OPT:
  → FIFO uses the time when a page was brought into memory.
  → OPT uses the time when a page is to be used.
- LRU Working principle:
  Replace the page that has not been used for the longest period of time.
- Each page is associated with the time of that page's last use

. • Example: Consider the following reference string:



- The first five faults are the same as those for optimal replacement.
- When the reference to page 4 occurs, LRU sees that of the three frames, page 2 was used least recently.
- Thus, the LRU replaces page 2.
- The LRU algorithm produces twelve faults.

• **S - Reference String**
• **S^R - Reverse of Reference String**
➢ **No of Page Faults of "S" using LRU = No of Page Faults of "S^R" using LRU**

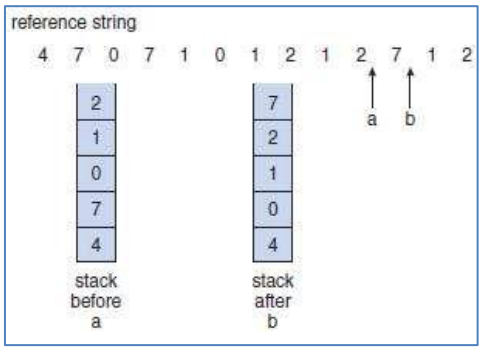
• **Two methods of implementing LRU:**

1) Counters

- Each page-table entry is associated with a time-of-use field.
- A counter(or logical clock) is added to the CPU.
- The clock is incremented for every memory-reference.
- Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- We replace the page with the smallest time value.

2) Stack

- Keep a stack of page-numbers
- Whenever a page is referenced, the page is removed from the stack and put on the top.
- The most recently used page is always at the top of the stack.
- The least recently used page is always at the bottom.
- This is illustrated in figure:
- Stack is best implement by a doubly linked-list.



• **Advantage:**

- Does not suffer from Belady's anomaly.

• **Disadvantage:**

-Few computer systems do not provide sufficient h/w support for true LRU page replacement.

• **Stack algorithms**

-Both LRU & OPT are called stack algorithms as they do not exhibit Belady's anomaly

-A stack algorithm is an algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames.

4.10.5 LRU-Approximation Page Replacement

• Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used.
• Many systems provide some help, however, in the form of a reference bit.
• The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page).
• Reference bits are associated with each entry in the page table.
• Initially, all bits are cleared (to 0) by the OS.
• As a user-process executes, the bit associated with each page referenced is set (to 1) by the hardware.
• By examining the reference bits, we can determine
→ which pages have been used and
→ which have not been used.
• This information is the basis for many page-replacement algorithms that approximate LRU replacement.

4.10.5.1 Additional-Reference-Bits Algorithm

• additional ordering information can be obtained by recording the reference bits at regular intervals.
• A 8-bit byte is used for each page in a table in memory.
• At regular intervals, a timer-interrupt transfers control to the OS.
• The OS shifts the reference bit for each page into the high-order bit of its 8-bit byte.
• These 8-bit shift registers contain the history of page use, for the last eight time periods.
• Examples:

00000000 - This page has not been used in the last 8 time units (800 ms).
11111111 - Page has been used every time unit in the past 8 time units.
11000100 (196) has been used more recently than 01110111(119).

• The page with the lowest number is the LRU page, and it can be replaced.
• the numbers are not guaranteed to be unique, however.
• We can either replace (swap out) all pages with the smallest value or use the FIFO method to choose among them.

4.10.5.2 Second-Chance Algorithm

• The number of bits of history included in the shift register can be varied to make the updating as fast as possible.
• In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the second-chance algorithm.
• Basic algorithm is a FIFO replacement algorithm.

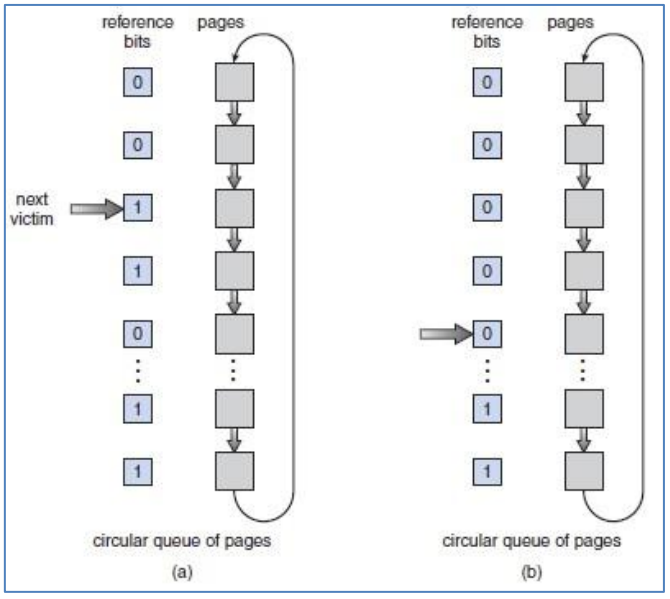
Procedure:

-When a page has been selected, we inspect its reference bit.
- If reference bit=0, we proceed to replace this page.
- If reference bit=1, we give the page a second chance & move on to select next FIFO page.
- When a page gets a second chance, its reference bit is cleared, and its arrival time is reset.

Implementation:

• A circular queue can be used to implement the second-chance algorithm(sometimes referred to as the *clock* algorithm) as shown in figure .

- A pointer (that is, a hand on the clock) indicates which page is to be replaced next.
- When a frame is needed, the pointer advances until it finds a page with a 0 reference bit.
- As it advances, it clears the reference bits.
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.



• **worst case:** when all bits are set, the pointer cycles through the whole queue, giving each page a second chance.
• It clears all the reference bits before selecting the next page for replacement.
• Second-chance replacement degenerates to FIFO replacement if all bits are set.

4.10.5.3 Enhanced Second-Chance Algorithm

• We can enhance the second-chance algorithm by considering
  1) Reference bit and
  2) modify-bit.
• We have following 4 possible classes:
  1) (0, 0) neither recently used nor modified -best page to replace.
  2) (0, 1) not recently used hut modified-not quite as good, because the page will need to be written out before replacement.
  3) (1, 0) recently used but clean-probably will be used again soon.
  4) (1, 1) recently used and modified -probably will be used again soon, and the page will be need to be written out to disk before it can be replaced.
• Each page is in one of these four classes.
• When page replacement is called for, we examine the class to which that page belongs.
• We replace the first page encountered in the lowest nonempty class.
• Notice that we may have to scan the circular queue several times before we find a page to be replaced.
• The major difference between this algorithm and the simpler clock algorithm is that here we give preference to those pages that have been modified to reduce the number of 1/Os required.

4.10.6 Counting-Based Page Replacement

1) LFU(Least Frequently Used) page-replacement algorithm

• **Working principle:** The page with the smallest count will be replaced.
-The reason for this selection is that an actively used page should have a large reference count.
• **Problem:** When a page is used heavily during initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.
• **Solution:**
Shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

2) MFU (Most Frequently Used) page-replacement algorithm

Working principle:
The page with the smallest count was probably just brought in and has yet to be used.

4.10.7 Page-Buffering Algorithms

Other procedures are often used in addition to a specific page-replacement algorithm,.

➢ systems commonly keep a pool of free frames.
  • When a page fault occurs, a victim frame is chosen as before.
  • However, the desired page is read into a free frame from the pool before the victim is written out.
  • This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out.
  • When the victim is later written out, its frame is added to the free-frame pool.

- to maintain a list of modified pages.
  - Whenever the paging device is idle, a modified page is selected and is written to the disk.
  - Its modify bit is then reset.
  - This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

- to keep a pool of free frames but to remember which page was in each frame.
  - Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused.
  - No I/O is needed in this case.
  - When a page fault occurs, we first check whether the desired page is in the free-frame pool, if it is not, we must select a free frame and read into it.
  - This technique is used in the VAX/VMS system along with a FIFO replacement algorithm.

4.10.8 Applications and Page Replacement

• In certain cases, applications accessing data through the operating system's virtual memory perform, worse than if the operating system provided no buffering at all.

- a database provides its own memory management and I/O buffering.

Applications like this understand their memory use and disk use better than does an operating system that is implementing algorithms for general-purpose use.

If the operating system is buffering I/O, and the application is doing so as well, then twice the memory is being used for a set of I/O.

- data warehouses frequently perform massive sequential disk reads, followed by computations and writes.

The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer.

Here, MFU would actually be more efficient than LRU.

• Because of such problems, some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures.

• This array is sometimes called the **raw disk**, and I/O to this array is termed **raw I/O.**

• Raw I/O bypasses all the file system services, such as file I/O demand paging, file locking, prefetchmg, space allocation, file names, and directories.

• Note that although certain applications are more efficient when implementing their own special-purpose storage services on a raw partition, most applications perform better when they use the regular file-system services.

4.11 Allocation of Frames

4.11.1 Minimum Number of Frames

• We cannot, for example, allocate more than the total number of available frames (unless there is page sharing).

• We must also allocate at least a minimum number of frames.

One reason for this is performance.

• As the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.

• In addition, when a page-fault occurs before an executing instruction is complete, the instruction must be restarted.

• The minimum number of frames is defined by the computer architecture.
• the maximum number is defined by the amount of available physical memory.

4.11.2 Allocation Algorithms

1) Equal Allocation

• We split m frames among n processes is to give everyone an equal share, m/n frames.

For ex: if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool.

2) Proportional Allocation

• various processes will need differing amounts of memory.

Problem:

• Consider a system with a 1-KB frame size.
• If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames.
• The student process does not need more than 10 frames, so the other 21 are, wasted.

Solution : we can use proportional allocation, in which we allocate available memory to each process according to its size.

• Let the size of the virtual memory for process *pi* be *si*, and define

$$S = \sum si$$

• Then, if the total number of available frames is *m*, we allocate *ai* frames to process pi, where *ai*, is approximately

$$ai = si/S \text{ X } m.$$

• Therefore for the above process the allocation will be as below:

s_i = size of process p_i	$m = 64$
$S = \sum s_i$	$s_1 = 10$
m = total number of frames	$s_2 = 127$
a_i = allocation for $p_i = \frac{s_i}{S} \times m$	$a_1 = \frac{10}{137} \times 64 \approx 5$
	$a_2 = \frac{127}{137} \times 64 \approx 59$

➢ In both equal and proportional allocation, the allocation may vary according to the multiprogramming level.

• If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process.
• Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

➢ Notice that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process.

• To give priority for a high-priority process a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority can be used

4.11.3 Global versus Local Allocation

• its Another important factor in the way frames are allocated to the various processes is page replacement.

Global Replacement	Local Replacement
Allows a process to a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.	Each process selects from only its own set of allocated frames.
A process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it.	Number of frames allocated to a process does not change.
The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes.	The set of pages in memory for a process is affected by the paging behavior of only that process.
Disadvantage: A process cannot control its own page-fault rate.	**Disadvantage:** Might prevent a process by not making available to it other less used pages of memory.
Advantage: Results in greater system throughput.	**Advantage:** More consistent per-process performance

4.12 Thrashing

• Minimum No of Frames must be allocated to Process to have Minimum No of Page Faults .

• If Process is not given minimum no of frames , it will quickly page-fault.

• At this point, it must replace some page.

• Since all its pages are in active use, it must replace a page that will be needed again right away.

• Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately

Thus, if Process is not given minimum no of frames,then Process spends most of its time in doing I/O than execution.

• This high paging activity is called thrashing.

• A process is thrashing if it is spending more time paging than executing.

4.12.1 Cause of Thrashing

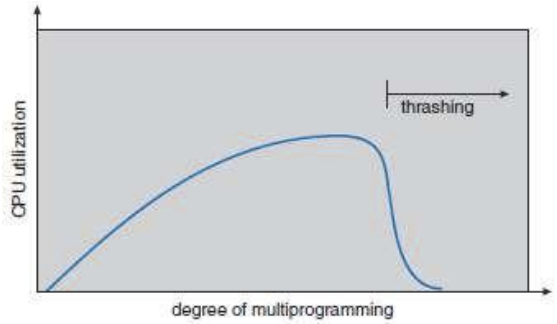
• Thrashing results in severe performance-problems

• The thrashing phenomenon:

- As processes keep faulting, they queue up for the paging device, so CPU utilization decreases
- The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result.
- The new process causes even more page-faults and a longer queue!

• As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.
• Thrashing has occurred, and system throughput plunges.
• The pagefault rate increases tremendously
• As a result, the effective memory-access time increases.
• No work is getting done, because the processes are spending all their time paging.

• This phenomenon is illustrated in Figure , in which CPU utilization is plotted against the degree of multiprogramming.
• As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached.
• If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
• At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.



• **Methods to avoid thrashing:**

1) Use Local Replacement

If one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

2) We must provide a process with as many frames as it needs. This approach defines the locality model of process execution.

• Locality Model states that

As a process executes, it moves from locality to locality.

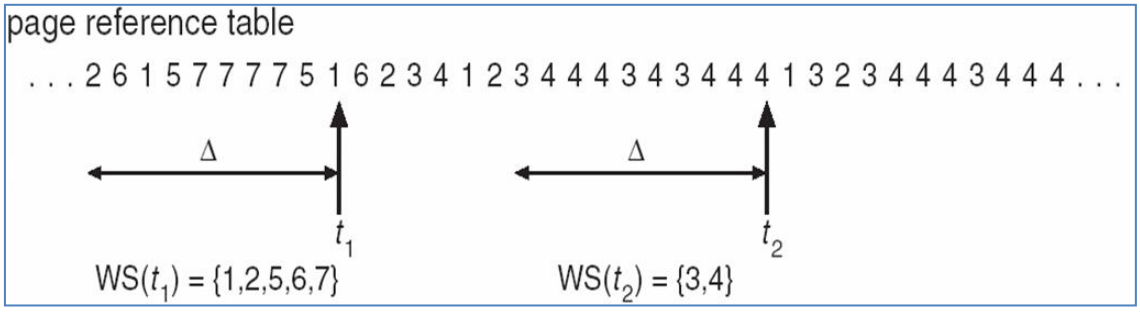
- A locality is a set of pages that are actively used together.
- A program may consist of several different localities, which may overlap.

4.12.2 Working-Set Model

• the working-set model is based on the assumption of locality.
• This model uses a parameter, Δ, to define the working-set window.
• The idea is to examine the most recent Δ page references.
• The set of pages in the most recent Δ page references is the working set (as shown in figure below).
• If a page is in,active use, it will be in the working set.
• If it is no longer being used, it will drop from the working set Δ time units after its last reference.
• Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure below, if Δ = 10 memory references, -then the working set at time *t1* is {1, *2,* 5, 6, *7).*

- By time t2, the working set has changed to {3, 4}.



• **Selection of Δ:**

-The accuracy of the working set depends on the selection of Δ.

-If Δ is too small, it will not encompass the entire locality;

-if Δ is too large, it may overlap several localities.

-In the extreme, if Δ is infinite, the working set is the set of pages touched during the process execution.

-Thus ,the most important property of the working set, is its size.

-If we compute the working-set size, *WSSi*, for each process pi in the system, we can then consider that

$$D=\sum WSSi$$

-where *D* is the total demand for frames.

-If the total demand is greater than the total number of available frames (D > *m)*, thrashing will occur, because some processes will not have enough frames.

• **Working principle:**

-Once Δ has been selected,

-The operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working-set size.

-If there are enough extra frames, another process can be initiated.

-If the sum of the working-set sizes increases, exceeding the total number of available frames, the operating -system selects a process to suspend.

-The process's pages are written out (swapped), and its frames are reallocated to other processes.

-The suspended process can be restarted later.

-This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.

-Thus, it optimizes CPU utilization.

• **Disadvantage:**

-The difficulty with the working-set model is keeping track of the working set.

-The working-set window is a moving window.

-At each memory reference, a new reference appears at one end and the oldest reference drops off the other end.

-A page is in the working set if it is referenced anywhere in the working-set window.

4.12.3 Page-Fault Frequency

• the page-fault frequency (PFF) takes a more direct approach.

• The specific problem is how to prevent thrashing.

• Thrashing has a high page-fault rate.

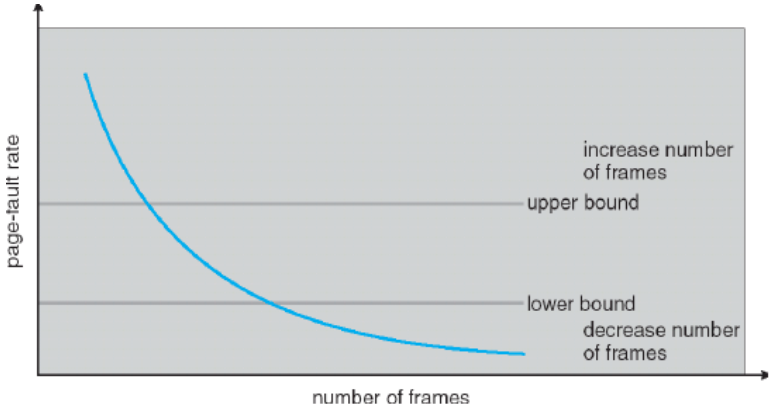
• Thus, we want to control the page-fault rate.

• When it is too high, we know that the process needs more frames.

• Conversely, if the page-fault rate is too low, then the process may have too many frames.

• Thus upper and lower bounds on the desired page-fault rate is established as shown in Figure below

• If the actual page-fault rate exceeds the upper limit, we allocate the process another frame;
• if the page-fault rate falls below the lower limit, we remove a frame from the process.
• Thus, we can directly measure and control the page-fault rate to prevent thrashing.
• If the page-fault rate increases and no free frames are available, we must select some process and suspend it.
• The freed frames are then distributed to processes with high page-fault rates.



MODULE 5
FILE SYSTEM, IMPLEMENTATION OF FILE SYSTEM

5.1 File Concepts

• A file is a named collection of related information on secondary-storage.

• Commonly, file represents

→ program (both source and object forms)and data.

• Data in file may be numeric , alphabetic or binary.

• A file has a certain defined structure, which depends on its type.

• Four types of file:

1) Text file: sequence of characters organized into lines.

2) Source file: sequence of subroutines & functions.

3) Object file: sequence of bytes organized into blocks.

4) Executable file: series of code sections.

• In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

5.1.1 File Attributes

• A file's attributes vary from one operating system to another but typically consist of these:

1) Name
- The only information kept in human-readable form.

2) Identifier
- It is a unique number which identifies the file within file-system.
- It is in non-human-readable form.

3) Type
- It is used to identify different types of files.

4) Location
- It is a pointer to
→ device and
→ location of file on that device.

5) Size
- Current-size of file in terms of bytes, words, or blocks.
- It also includes maximum allowed size.

6) Protection
- Access-control information determines who can do reading , writing and executing.

7) Time, date, & user identification
- These information can be kept for
→ creation
→ last modification and
→ last use.
- These data can be useful for
→ protection
→ security and
→ usage monitoring.

• Information about files are kept in the **directory-structure,** which is maintained on the disk.

• A directory entry consists of **the file's name and its unique identifier.**

• The identifier in turn locates the other file attributes.

• It may take more than a kilobyte to record this information for each. file.

5.1.2 File Operations

Following are six basic file operations:

1) Creating a file

- Two steps are:
  - i) Find the space in the file-system for the file.
  - ii) An entry for the new file is made in the directory.

2**) Writing a file**

- Make a system-call specifying both
  - → file-name and
  - → information. to be written to the file.
- The system searches the directory to find the file's location.
- The system keeps a writepointer(wp) to the location in the file where the next write is to take place)
- The write-pointer must be updated whenever a write-operation occurs.

3) Reading a file

- Make a system-call specifying both
  - → file-name and
  - → location of the next block of the file in the memory.
- The system searches the directory to find the file's location.
- The system keeps a readpointer(rp) to the location in the file where the next read is to take place.
- The read-pointer must be updated whenever a read-operation occurs.
- Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process current file- position pointer
- Both the read and write operations use this same pointer, saving space and reducing system complexity.

4) Repositioning within a file

Two steps are:

i) Search the directory for the appropriate entry.

ii) Set the current-file-position to a given value.

- This file-operation is also known as file seek.

5) Deleting a file

- Two steps are:
  - i) Search the directory for the named-file.
  - ii) Release all file-space and erase the directory-entry.

6) Truncating a file

- The contents of a file are erased but its attributes remain unchanged.
- Only file-length attribute is set to zero.

• These primitive operations can then be combined to perform other file operations.

• Most of the above file-operations involve searching the directory for the entry associated with the file.

• To avoid this constant searching, many systems require that an open system-call be used before that file is first used

• The OS keeps a small table which contains information about all open files called open-file table.

• If a file-operation is requested, then
→ file is specified via an index into open-file table
→ so no searching is required.
• If the file is no longer actively used, then
→ process closes the file and
→ OS removes its entry in the open-file table.
• Two levels of internal tables:

1) Per-process Table

- Tracks all files that a process had opened.
- Includes access-rights to
→ file and
→ accounting information.
- Each entry in the table in turn points to a system-wide table

2) System-wide Table

- Contains process-independent information. such as
→ file-location on the disk
→ file-size and
→ access-dates.

• Information associated with an open file:

1) File-pointer

- Used by the system to keep track of last read-write location.

2) File-open Count

- The counter
→ tracks the no. of opens & closes and
→ reaches zero on the last close.The system can then remove the entry in open-file table entries and reuse that space.

3) Disk Location of the File

- Location-info is kept in memory to avoid having to read it from disk for each operation.

4) Access Rights

- Each process opens a file in an access-mode (read, write or execute).
- This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

• File locks allow one process to
→ lock a file and
→ prevent other processes from gaining access to locked-file.

Shared Lock	Exclusive Lock
Similar to a reader lock.	Behaves like a writer lock.
Several processes can acquire the lock concurrently.	Only one process at a time can acquire the lock.

• operating systems may provide either mandatory or advisory file-locking mechanisms.

Mandatory	Advisory
OS will prevent any other process from accessing the locked-file.	OS will not prevent other process from accessing the locked-file.
OS ensures locking integrity.	It is up to software-developers to ensure that locks are appropriately acquired and released.
Used by windows OS.	Used by UNIX systems.

• The use of file locks requires the same precautions as ordinary process synchronization

5.1.3 File Types

• When we design a file system we always consider whether the operating system should recognize and support file types.

• If an operating system recognizes the type of a file, it can then operate on the file

• Common technique for implementing file-types: **Include the type as part of the file-name**.

• Two parts of file-name:

1) Name and 2) Extension

• The system uses the extension to indicate

→ type of file and

→ type of operations (read or write).

• Example:

→ Only a file with a .com, .exe, or .bat extension can be executed.

→ .com and .exe are two forms of binary executable files.

→ .bat file is a batch file containing, in ASCII format, commands to the OS.

•table lists different types of file and their extentions

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

5.1.4 File Structure

• File types can be used to indicate the internal structure of the file.

• Disadvantage of supporting multiple file structures: os size will become very large.

• All OSs must support at least one structure: an executable file

• Some operating systems impose (and support) a minimal number of file structures.

• For example, in MAC OS,

-It expects files to contain two parts: a resource fork and a data fork.

1) Resource fork: contains information of interest to the user.

-For instance, it holds the labels of any buttons displayed by the program.

2) Data fork: contains program-code or data.

• Too few structures make programming inconvenient.

• Too many structures make programmer confusion.

5.1.5 Internal File Structure

• Locating an offset within a file can be complicated for the OS.
• Disk-systems typically have a well-defined block-size.
• All disk I/0 is performed in units of one block (physical record), and all blocks are the same size.
• **Problem:** It is unlikely that physical-record size will exactly match length of desired logical-record.
• **Solution:** Packing a number of logical-records into physical-blocks.
• Following things determine how many logical-records are in each physical-block:
→ logical-record size
→ physical-block size and
→ packing technique.
• The packing can be done either by
→ user's application program or
→ OS.
• Disadvantage of packing:
-Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted
- All file-systems suffer from internal fragmentation
- the larger the block size, the greater the internal fragmentation.

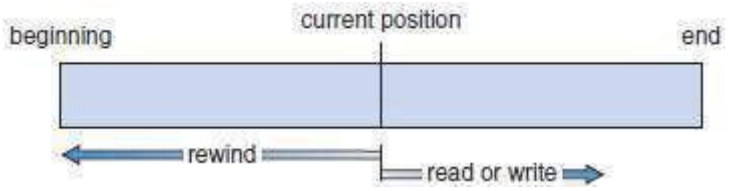
5.2 Access Methods

• Files store information.
• When it is used, this information must be accessed and read into computer memory.
• The information in the file can be accessed in several ways.

5.2.1 Sequential Access

• Information in the file is processed in order as shown in figure below.
• Reading and writing are the 2 main operations on the file.
• File-operations:
1) read next
-This is used to
→ read the next portion of the file and
→ advance a file-pointer, which tracks the I/O location.
2) write next
- This is used to
→ append to the end of the file and
→ advance to the new end of file.

• Sequential access, which is depicted in Figure is based on a tape model of a file.
• This works both on
→ sequential-access devices and
→ random-access devices.



5.2.2 Direct Access (Relative Access)

• This is based on a disk model of a file (since disks allow random access to any file-block).
• A file is made up of fixed length logical records.
• Programs can read and write records rapidly in no particular order.
• **Advantages:**
1) Useful for immediate access to large amounts of information
2) Databases are often of this type.
• File-operations include a relative block-number as parameter.
• The relative block-number is an index relative to the beginning of the file.
• the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.
• Use of relative block-numbers:
→ allows OS to decide where the file should be placed and
→ helps to prevent user from accessing portions of file-system that may not be part of his file.
• File-operations:
1) read n
2) write n
-where n is the block-number
• Assuming we have a logical record length *L*,
• the request for record n is turned into an I/O request for *L* bytes starting at location *L* * (n) within the file (assuming the first record is n = 0).
• Since logical records are of a fixed size, it is also easy to read, write, or delete a record.
• Not all operating systems support both sequential and direct access for files.

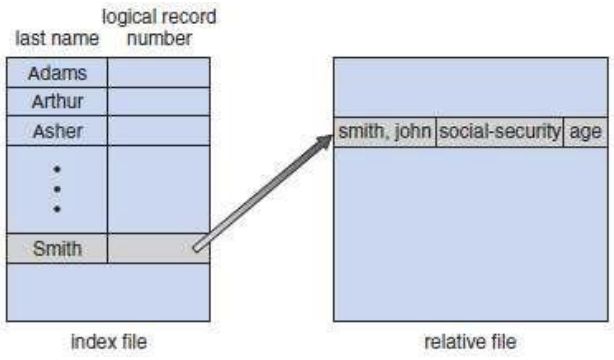
• We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as shown in Figure .

sequential access	implementation for direct access
reset	cp = 0;
read_next	read cp ; cp = cp + 1;
write_next	write cp; cp = cp + 1;

• Simulating a direct-access file on a sequential-access file, however, is extremely inefficient and clumsy.

5.2.3 Other Access Methods

• Other access methods can be built on top of a direct-access method.
• These methods generally involve constructing a index for a file.
• The index contains pointers to the various blocks (like an index in the back of a book).
• To find a record in the file:
1) First, search the index and
2) Then, use the pointer to
→ access the file directly and
→ find the desired record.
Figure shows example of index and relative files



• Problem: With large files, the index-file itself may become too large to be kept in memory.
• Solution: Create an index for the index-file. (The primary index-file may contain pointers to secondary index-files, which would point to the actual data-items).

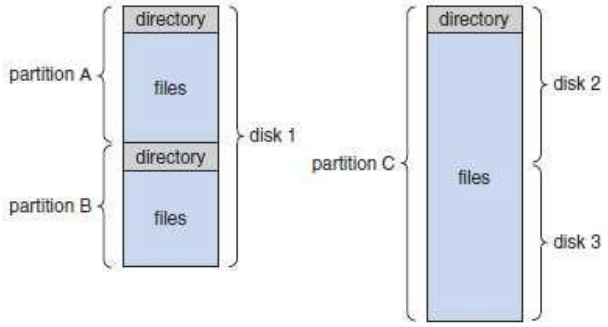
5.3 Directory Structure

• The file systems of computers, can be extensive.
• Some systems store millions of files on terabytes of disk.
• To manage all these data, we need to organize them. This organization involves the use of directories.

5.3.1 Storage Structure

• A storage-device can be used in its entirety for a file-system.
• The storage-device can be split into 1 or more partitions (known as slices or minidisk).
• a chunk of storage that holds a file system is reffered as a volume.
• Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one.
• The volume may be
→ a subset of a device or
→ a whole device.
• Each volume must also contain information about the files in the system. This information is kept in entries in a device directory(or volume table of contents).
• Device directory (or directory) records following information for all files on that volume:
→ name
→ location
→ size and
→ type.

• Figure below shows a typical file-system organization.



5.3.2 Directory Overview

• Operations performed on a directory:

1) Search for a File
- We need to be able to search a directory-structure to find the entry for a particular file.

2) Create a File
- We need to be able to create and add new files to the directory.

3) Delete a File
- When a file is no longer needed, we want to be able to remove it from the directory.

4) List a Directory
- We need to be able to
→ list the files in a directory and
→ list the contents of the directory-entry for each file.

5) Rename a File
- Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.
- Renaming a file may also allow its position within the directory structure to be changed.

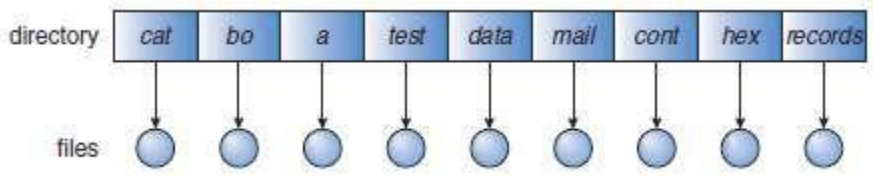
6) Traverse the File-system

- We may wish to access
  → every directory and
  → every file within a directory-structure.
- For reliability, it is a good idea to save the contents and structure of the entire file-system at regular intervals.

- The most common schemes for defining the logical structure of a directory.
  1) Single level directory
  2) Two level directory
  3) Tree structured directories
  4) Acyclic-graph directories
  5) General graph directory

5.3.3 Single Level Directory

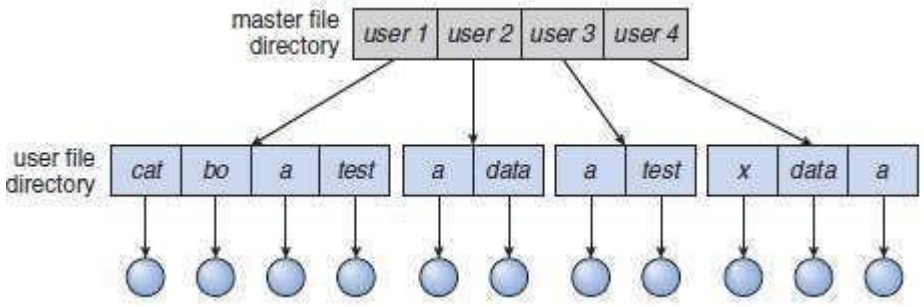
- It is the simplest directory structure.
- All files are contained in the same directory (as shown Figure).



- Disadvantages (Limitations):
  - All files must have unique names.
  - Difficult to remember names of all files, as number of files increases.

5.3.4 Two Level Directory

- A separate directory for each user.
- Each user has his own UFD (user file directory).
- The UFDs have similar structures.
- Each UFD lists only the files of a single user.
- When a user job starts, the system's MFD is searched (MFD=master file directory).
- The MFD is indexed by user-name or account number.
- Each entry in MFD points to the UFD for that user (as shown in Figure).

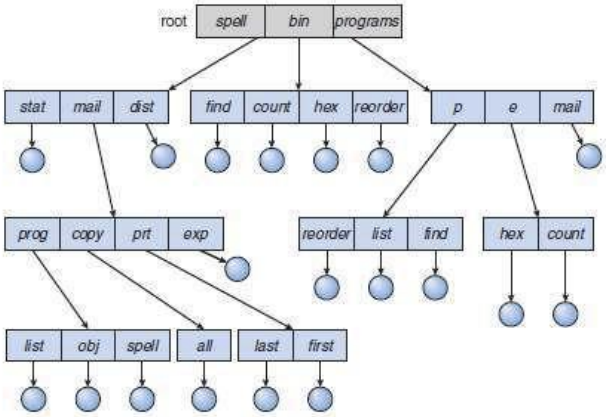


• Different users may have files with the same name, as long as all the file names within each UFD are unique
• To create a file for a user,
-the OS searches only that user's UFD to determine whether another file of that name exists.
• To delete a file,
-the OS limits its search to the local UFD. (Thus, it cannot accidentally delete another user's file that has the same name).
• Advantages:
1) No filename-collision among different users.
2) Efficient searching.
• Disadvantage:
1) Users are isolated from one another and can't cooperate on some task and to access one another's files.
• **Solution:**
• If access is to be permitted, one user must have the ability to name a file in another user's directory.
• To name a particular file uniquely in a two-level directory, we must give both the user name and the file name.
• A two-level directory can be thought of as a tree, or an inverted tree, of height 2.
-The root of the tree is the MFD.
-Its direct descendants are the UFDs.
-The descendants of the UFDs are the files themselves.
-The files are the leaves of the tree.
• Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).
• Thus, a user name and a file name define a pathname.
• Every file in the system has a path name.
• To name a file uniquely, a user must know the path name of the file desired.
• For example,
-if user A wishes to access her own test file named test, she can simply refer to test.
-To access the file named test of user B (with directory-entry name userb), however, she might have to refer to /userb/test.

• Every system has its own syntax for naming files in directories other than the user's own.
• Additional syntax is needed to **specify the volume of a file**.
• For instance, in MS-DOS a volume is specified by a letter followed by a colon.
-Thus, a file specification might be C:\ userb\test
• Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files.
• When the appropriate commands are given to the operating system, these files are read by the loader and executed.
• A **special user directory** is defined to contain the **system files** (for example, user 0).
• Whenever a file name is given to be loaded, the operating system first searches the local UFD.
• If the file is found, it is used. If it is not found, the system automatically searches the special user directory that contains the system files.
• The sequence of directories searched when a file is named is called the **search path**.
• The search path can be extended to contain **an unlimited list of directories** to search when a command name is given.

5.3.5 Tree Structured Directories

• Users can create their own subdirectories and organize files (as shown in figure).
• A tree is the most common directory-structure.
• The tree has a root directory.
• Every file in the system has a **unique path-name**.
• A directory contains a set of files (or subdirectories).
• A directory is simply another file, but it is treated in a special way.
• In each directory-entry, one bit defines as
file (0) or
subdirectory (1).
• each process has a current directory.
• The **current directory** should contain most of the files that are of current interest to the process
• **Path-names can be of 2 types:**
1) **Absolute path-name** begins at the root and follows a path down to the specified file, giving the directory names on the path
2**) Relative path-name** defines a path from the current directory.



• For example, in the tree-structured file system in the above Figure , if the current directory is root/spell/mail, then **the relative path** name **prt/first** refers to the same file as does the **absolute path** name **root/spcll/mail/prt/first**.

• How to delete directory?
1) To delete an empty directory:
→ Just delete the directory.
2) To delete a non-empty directory:
Solution 1:→ First, delete all files in the directory.
→ If any subdirectories exist, this procedure must be applied recursively to them.
Solution2: When a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted.
• This policy is more convenient, but it is also more dangerous, because an entire directory structure can be removed with one command ex:rm command in UNIX
• Advantage:
1) Users can be **allowed to access the files of other** users by specifying proper path names.
• Disadvantages:
1) A path to a file can be longer than a path in a two-level directory.
2) Prohibits **the sharing of files** (or directories).

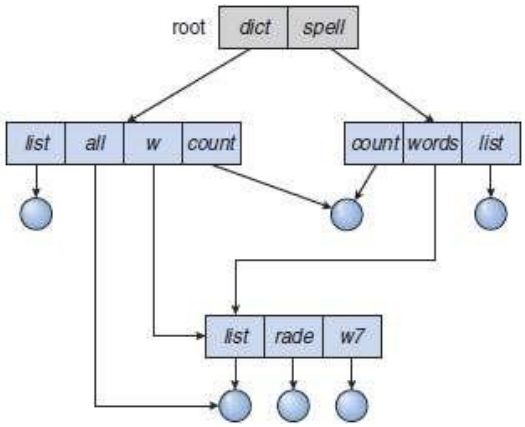
5.3.6 Acyclic Graph Directories

• An acyclic graph —that is, **a graph with no cycles**—allows directories to share subdirectories and files (as shown in Figure below).

• The same file or subdirectory may be in two different directories.

• The acyclic graph is **a natural generalization** of the tree-structured directory scheme.

• Only **one shared-file exists**, so any changes made by one person are immediately visible to the other.



• methods to implement shared-files(or subdirectories):

1) Create a new directory-entry called a link.

-A link is a pointer to another file (or subdirectory).

- Links are resolved by using that path name to locate the real file

2) Duplicate all information about shared-files in both sharing directories.

-A major problem with duplicate directory entries is maintaining consistency when a file is modified.

• **problems:**

1) A file may have **multiple absolute path-names.**

2) Deletion may leave **dangling-pointers** to the non-existent file.

• **Solution to deletion problem:**

1) Preserve the file until all references to it are deleted.

-a list of all references to a file is kept .

-When a link or a copy of the directory entry is established, a new entry is added to the file reference list.

-When a link or directory entry is deleted, we remove its entry on the list.

The file is deleted when its file-reference list is empty.

2) With symbolic links, remove only the link, not the file. If the file itself is deleted, the link can be removed.

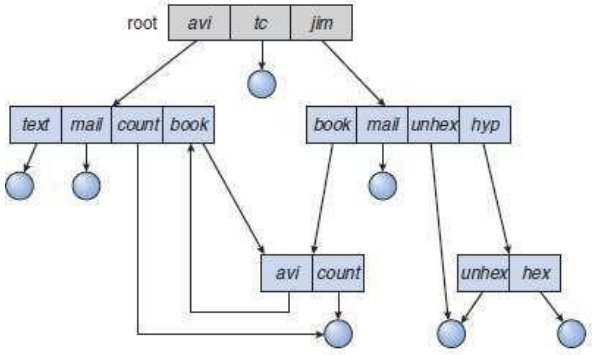
5.3.7 General Graph Directory

• when we **add links to an existing tree-structured directory**, the tree structure is destroyed, resulting in a simple graph structure

• If we have cycles,then,

-If we have just searched a major shared subdirectory for a particular file without finding it, we should avoid searching that subdirectory again as the second search would be a waste of time

• A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating



• **Solution1:** Limit the no. of directories accessed in a search.

• **Problem:** With cycles, the reference-count may be non-zero even when it is no longer possible to refer to a directory (or file).

• **Solution2: Garbage-collection** scheme can be used to determine when the last reference has been deleted.

• Garbage collection involves

1) First pass

→ traverses the entire file-system and

→ marks everything that can be accessed.

2) A second pass collects everything that is not marked onto a list of free-space.

• Disadvantage:

-Garbage collection is **necessary only because of possible cycles in the graph**.

-Thus, an acyclic-graph structure is much easier to work with.

-The difficulty is to avoid cycles as new links are added to the structure.

-There are algorithms to detect cycles in graphs; however, they are computationally expensive, especially when the graph is on disk storage.

• **Solution3:**

- A simpler algorithm in the special case of directories and links is **to bypass links during directory traversal**.

- Cycles are avoided, and no extra overhead is incurred.

5.4 File System Mounting

• A file-system must be mounted before it can be available to processes on the system

• Mount-point is the location in the file-structure where the file-system is to be attached.

a mount point is an empty directory.

• Procedure:

1) OS is given

→ name of the device and

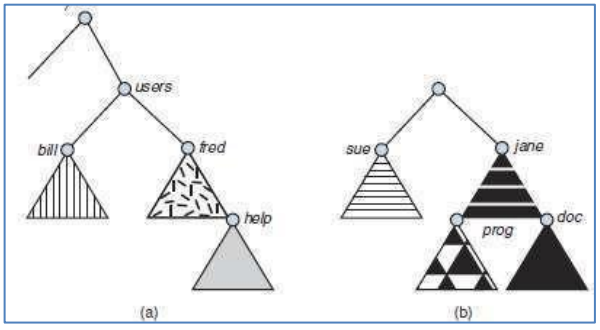
→ mount-point

2) OS verifies that the device contains a valid file-system.

3) OS notes in its directory-structure that a file-system is mounted at specified mount-point.

• This scheme enables the operating system to traverse its directory structure.

• To illustrate file mounting, consider the file system depicted in Figure below, where the triangles represent subtrees of directories that are of interest.

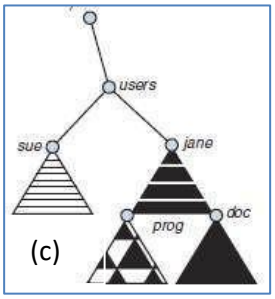


• Figure (a) shows an existing file system, while Figure(b) shows an unmounted volume residing on /device/disk.

• At this point, only the files on the existing file system can be accessed.

Figure (c) shows the effects of mounting the volume residing on /device/dsk over /users.

• If the volume is unmounted, the file system is restored to the situation depicted in Figure(a) abd (b).



• Systems impose semantics to clarify functionality.

• For example,

-a system may disallow a mount over a directory that contains files;

- a system may allow the same file system to be mounted repeatedly, at different mount points; or it may only allow one mount per file system.

5.5 File Sharing

• Sharing of files on multi-user systems is desirable.
• Sharing may be done through **a protection scheme**.
• On distributed systems, files may be shared across a network.
• Network File-system (NFS) is a common distributed file-sharing method.

5.5.1 Multiple Users

• Given a directory structure that allows files to be shared by users, the system must mediate the file sharing.
• File-sharing can be done in **2 ways:**

1) The system can allow a user to access the files of other users **by default** or
2) The system may require that a user **specifically grant access**.

• To implement file-sharing, the system must maintain **more file- & directory-attributes** than on a single-user system.
• Most systems use concepts of file **owner and group.**

1) Owner

-The user who

→ may change attributes & grant access and
→ has the most control over the file (or directory).

-Most systems implement owner attributes by managing a list of user-names and user IDs

2) Group

- The group attribute defines a subset of users who can share access to the file.
- Group functionality can be implemented as a system-wide list of group-names and group IDs.

• Exactly **which operations** can be executed by group-members and other users is definable by the file's owner.
• The owner and group IDs of a file

→ are **stored** with the other file-attributes.
→ can be used to **allow/deny** requested operations.

5.5.2 Remote File Systems

• Allows a computer to mount 1 or more file-systems from 1 or more remote-machines.
• Three methods:

1) Manually transferring files between machines **via programs like FTP**.
2)Automatically **DFS (Distributed file-system):** remote directories are visible from a local machine.
3) Semi-automatically via www **(World Wide Web):** A browser is needed to gain access to the remote files, and separate operations (a wrapper for ftp) are used to transfer files.

• **ftp is used for both anonymous and authenticated access.**
• **Anonymous access** allows a user to transfer files without having an account on the remote system.

5.5.2.1 Client Server Model

• Allows clients to mount remote file-systems from servers.
• The machine containing the files is called the server.
• The machine seeking access to the files is called the client.

• A server can serve multiple clients, and A client can use multiple servers.
• The server specifies which resources (files) are available to which clients.
• A client can be specified by a network-name such as an IP address.
• Disadvantage:
1) Client identification is more difficult.
• In UNIX and its NFS (network file-system), authentication takes place via the client networking information, by default.
• Once the remote file-system is mounted, file-operation requests are sent to the server via the DFS protocol.

5.5.2.2 Distributed Information Systems

• Provides **unified access** to the information needed for remote computing.
• The DNS (domain name system) provides hostname-to-networkaddress translations.
• Other distributed information systems provide username/password space for a distributed facility.
• **Sun Microsystems** introduced yellow pages (since renamed network information service, or NIS), and most of the industry adopted its use.
-It centralizes storage of user names, host names, printer information, and the like.
-Unfortunately, it uses unsecure authentication methods.
-Sun's NIS+ is a much more secure replacement for NIS but is also much more complicated
• In the case **of Microsoft, common internet file system** (CIFS), network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access to a requested file system.
-Microsoft uses two distributed naming structures to provide a single name space for users.
-The older naming technology is **domains.**
-The newer technology, available in Windows XP and Windows 2000, is **active directory.**
• the **lightweight directory-access protocol (LDAP)** a secure distributed naming mechanism.
-active directory is based on LDAP.
-Sun Microsystems includes LDAP with the operating system and allows it to be used for user authentication as well as system-wide retrieval of information,
• it better to have **one distributed LDAP** directory by an organization to store all user and resource information for all the organization's computers.
• The result would be **secure single sign-on for users**, who would enter their authentication information once for access to all computers within the organization.
• It would also **ease systems-administration**

5.5.2.3 Failure Modes

• **Local file-systems can fail** for a variety of reasons such as
→ failure of disk (containing the file-system)
→ corruption of directory-structure &
→ cable failure.
• Remote file-systems have more failure modes because of the **complexity of network-systems.**
• The network can be interrupted between 2 hosts. Such interruptions can result from
→ hardware failure
→ poor hardware configuration or
→ networking implementation issues.

• DFS protocols allow delaying of file-system operations to remote-hosts, with the hope that the remote-host will become available again.

- To implement failure-recovery, some kind of state information may be maintained on both the client and the server.

• NFS takes a simple approach, implementing a stateless DFS.

- In essence, it assumes that a client request for a file read or write would not have occurred unless the file system had been remotely mounted and the file had been previously open.

-The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation.

-Similarly, it does not track which clients have the exported volumes mounted, assuming that it must be legitimate.

• While this stateless approach makes NFS resilient and rather easy to implement, it also makes it unsecure.

• In NFS version 4, NFS is inade stateful to improve its security,

5.5.3 Consistency Semantics

• These represent an important **criterion of evaluating file-systems that supports file-sharing**.

• These **specify how multiple users of a system are to access a shared-file simultaneously**.

• In particular, they specify when modifications of data by one user will be observed by other users.

• These semantics are **typically implemented as code with the file-system**.

• These are directly related to the process-synchronization algorithms.

• A successful implementation of complex sharing semantics can be found in the Andrew file-system (AFS).

- **UNIX Semantics**

  - UNIX file-system (UFS) uses the following consistency semantics:

    1) Writes to an open-file by a user are **visible immediately to other** users who have this file opened.

    2) One mode of sharing allows users to **share the pointer of current location** into a file.

    Thus, the advancing of the pointer by one user affects all sharing users.

  -A file is associated with a **single physical image** that is accessed as an exclusive resource.

  -**Contention f**or the single image causes delays in user processes.

- **Session Semantics**

  - The AFS uses the following consistency semantics:

    1) Writes to an open file by a user **are not visible immediately to other users** that have the same file open.

    2) Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.

  -A file may be associated temporarily with **several (possibly different) images at the same time.**

  - Consequently, multiple users are allowed to perform both read and write accesses concurrently on their images of the file, **without delay.**

  - Almost **no constraints** are enforced on scheduling accesses.

- **Immutable Shared Files Semantics**

  -Once a file is declared as shared by its creator, it cannot be modified.

- An immutable file has 2 key properties:
  1**) File-name may not be reused and**
  2) File-contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed system is simple, because the sharing is disciplined

5.6 Protection

• When **information is stored in a computer system, we want to keep it safe from physical damage** (reliability) and improper access (protection).

• **File systems can be damaged** by
-hardware problems (such as errors in reading or writing),
-power surges or failures,
-head crashes, -dirt,
-temperature extremes, and vandalism.
-Files may be deleted accidentally.
-Bugs in the file-system software can also cause file contents to be lost.

• **Reliability** is generally provided by duplicate copies of files.

• File owner/creator should be able to control:
→ what can be done
→ by whom.

• For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer.

• In a multiuser system, however, other mechanisms are needed.

5.6.1 Types of Access

• **Systems that do not permit access to the files of other users do not need protection. Or We could provide free access with no protection.**

• These are too extreme, so controlled-access is needed.

• Following operations may be controlled:
1) Read
- Read from the file.
2) Write
- Write or rewrite the file.
3) Execute
- Load the file into memory and execute it.
4) Append
- Write new information at the end of the file.
5) Delete
- Delete the file and tree its space for possible reuse.
6) List
- List the name and attributes of the file.

• Other operations, such as renaming, copying, and editing the file, may also be controlled.

• For many systems, however, these higher-level functions may be implemented using lower-level system calls.

• **So protection is provided at only the lower level.**

5.6.2 Access Control

• Common approach to protection problem: **make access dependent on identity of user**.

• Files can be associated with an ACL (access-control list) which specifies

→ username and

→ types of access for each user.

• **Problems:**

1) Constructing a list can be tedious.

2) Directory-entry now needs to be of variable-size, resulting in more complicated space management.

• **Solution:** These problems can be resolved by **combining ACLs with an owner, group, universe access control scheme**

• To reduce the length of the ACL, many systems recognize 3 classifications of users:

1) Owner

-The user who created the file is the owner.

2) Group

- A set of users who are sharing the file and need similar access is a group.

3) Universe

- All other users in the system constitute the universe.

• With the more limited protection classification, only three fields are needed to define protection.

• each field is a collection of bits, and each bit either allows or prevents the access associated with it.

• For example,

-the **UNfX system defines three fields of 3 bits each—rwx,** where

-r controls read access,

-w controls write access, and

-x controls execution.

• A separate field is kept for the file owner, for the file's group, and for all other users.

• In this scheme, nine bits per file are needed to record protection information.

• For example, the protection fields for the owner all bits are set; for the group the r and w bits are set; and for the universe, only the r bit is set.

• Windows XP users typically manage access-control lists **via the GUI.**

5.6.3 Other Protection Approaches

• **A password** can be associated with each file.

• Disadvantages:

1) The no. of passwords you need to remember may become large.

2) If only one password is used for all the files, then all files are accessible if it is discovered.

• In a multilevel directory-structure, we need to **provide a mechanism for directory protection**.

• **The directory operations** that must be protected are **different from the File-operations**:

1) to Control creation & deletion of files in a directory.

2) to Control whether a user can determine the existence of a file in a directory,i.e, **listing the contents of a directory must be a protected operation**

3) In systems where files may have numerous path names (such as acyclic or general graphs), a given user may have different access rights to a particular file, depending on the path name used.

FILE-SYSTEM IMPLEMENTATION

5.7 File System Structure

• **Disks provide the bulk of secondary-storage** on which a file-system is maintained.

• The disk is a suitable medium for storing multiple files. This is because

1) A disk can **be rewritten** in place.

- It is possible to

→ read a block from the disk

→ modify the block and

→ write the block into the disk.

2) A disk can **access directly any block of information**.

- It is possible to access any file either sequentially or randomly.

- Switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

• To improve I/O efficiency, **I/O transfers between memory and disk are performed in units of blocks**.

-Each block has one or more sectors.

- Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes.

-The usual size is 512 bytes.

• File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily

• Design problems of file-systems:

1) Defining how the file-system should look to the user.

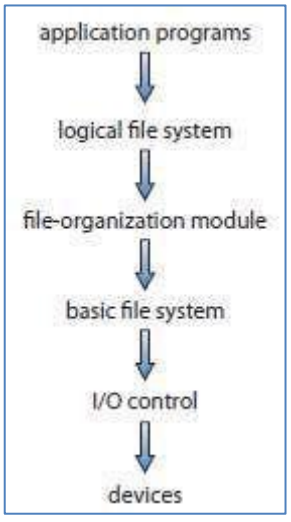
2) Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

• The **file-system itself is generally composed of many different levels**.

• Every level in design uses features of lower levels to create new features for use by higher levels.

Layered File System

• Levels of the file-system shown in Figure :



1) I/O Control (Lowest level)

-Consists of device-drivers & interrupt handlers to transfer information between main-memory & disk.

-A device-driver can be thought of as a translator.

-Its input consists of high-level commands.

-Its output consists of low-level instructions.

2) Basic File-system

- Needed only to **issue basic commands to the appropriate device-driver** to read & write blocks on the disk.

3) File-organization Module

- Knows about files and their **logical & physical blocks.**

- Translates **logical-block address to physical-block address.**

4) Logical File-system

- **Manages metadata information.** i.e. Metadata includes all of the file-system structure except the actual data. Manages the directory-structure.

- Maintains file-structure via FCB (File Control Blocks). i.e. **FCB** contains information about the file, including → ownership, permissions and location of the file.

• Advantages of layered structure:

1) Duplication of code is minimized.

2) I/O control can be used by multiple file-systems

5.8 File System Implementation

5.8.1 Overview

• On-disk & in-memory structures are used to implement a file-system.

• **On-disk structures include** :

1) Boot Control Block

- Contains information needed to boot an OS from the partition.
- It is typically the first block of a volume.
- In UFS, it is called the boot block.
-In NTFS, it is the partition boot sector.

2) volume Control Block

-Contains partition-details like

→ no. of blocks

→ size of blocks and

→ free-block count.

- In UFS, this is called a superblock.
-In NTFS, it is stored in the master file table.

3) Directory-structure

- Used to organize the files.
- In UFS, this includes file-names and associated inode-numbers.
- In NTFS, it is stored in the master file table.

4) FCB (file control block)(as shown in figure)

-Contains file-details(per file) including

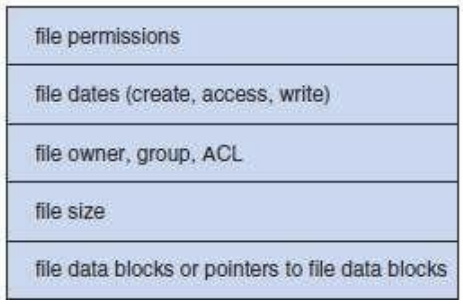
→ file-permissions

→ ownership

→ file-size and

→ location of data-blocks.

- In UFS, this is calledthe inode.
- In NTFS, this information is actually stored within the masterfile table


file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

• **In-memory structures** are used for both file-system management and performance improvement via caching:

The structures may include:

1) In-memory Mount Table

- Contains information about each mounted partition.

2) An in-memory Directory-structure cache

- Holds directory information of recently accessed directories.

3) System-wide Open-file Table

- Contains a copy of the FCB of each open file

4) Per-process Open-file Table

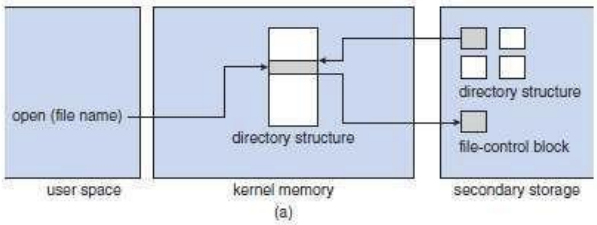
- Contains a pointer to the appropriate entry in the system-wide open-file table and some other information

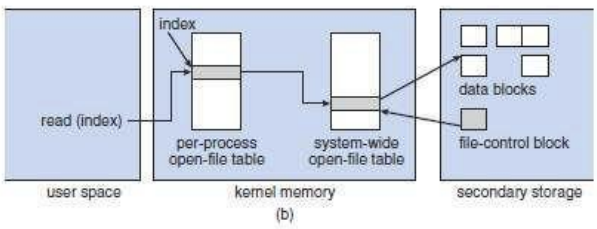
• Buffers hold file-system blocks when they are being read from disk or written to disk.
• To create a new file, a program calls the LFS (logical file-system).
• The LFS knows the format of the directory-structures.
• **To create a new file,**
- the LFS
1) Allocates a new FCB.
2) Reads the appropriate directory into memory.
3) Updates it with the new file-name and FCB.
4) Writes it back to the disk
• **To open a file**,
-After a file has been created, it can be used for I/O:
-First, it must be opened.
-The open( ) call passes a file name to the file system.
-The open( ) system call first searches the system-wide open-file table to see if the file is already in use by another process.
-If it is,
- a per-process open-file table entry is created pointing to the existing system-wide open-file table.
-Else,
- the directory structure is searched for the given file name.
- Once the file is found, the FCB is copied into a system-wide open-file table in memory.
- This table not only stores the FCB but also tracks the number of processes that have opened the file.
- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.
- The open() call returns a pointer to the appropriate entry in the per-process file-system table.
- All file operations are then performed via this pointer.
-UNIX systems refer to it as a **file descriptor;**
- Windows refers to it as a **file handle.**
• **To close a file**:
-When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented.
-When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

• The operating structures of a file-system implementation are summarized in Figure below.

Figure (a) illustrate File open.

Figure (b) illustrate File read





5.8.2 Partitions & Mounting

• Disk layouts can be:

1) A disk can be divided into multiple partitions or
2) A partition can span multiple disks (RAID).

• Each partition can either be:

1) Raw i.e. containing no file-system or
2) Cooked i.e. containing a file-system.

• **Boot information**

-Boot information can be stored in a separate partition

-The **boot information has its own format**, because

→ at boot time the system does not have device-drivers loaded and

→ the system cannot interpret the file-system format.

-Thus, **Boot information is a sequential series of blocks**, loaded as an image into memory.

-Execution of the image starts at a predefined location, such as the first byte.

• **Steps for mounting:**

1) The root partition containing the kernel is mounted at boot time.
2) Then, the OS verifies that the device contains a valid file-system.
   -If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.
3) Finally, the OS notes in its in-memory mount table structure that
   i) A file-system is mounted and
   ii) Type of the file-system.

• The details of this function depend on the operating system.

• Microsoft Windows-based systems mount each volume in a separate name space, denoted by a letter and a colon.

• On UNIX, file systems can be mounted at any directory.

- Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory.
-The flag indicates that the directory is a mount point.

5.8.3 Virtual File Systems

• **The OS allows multiple types of file-systems to be integrated into a directory-structure**.

• most operating systems, use object-oriented techniques to simplify, organize, and modularize the implementation.

• The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS.

• Users can access files that are contained within multiple file systems on the local disk or even on file systems available across the network.

• Data structures and procedures are used to isolate the basic system call functionality from the implementation details.

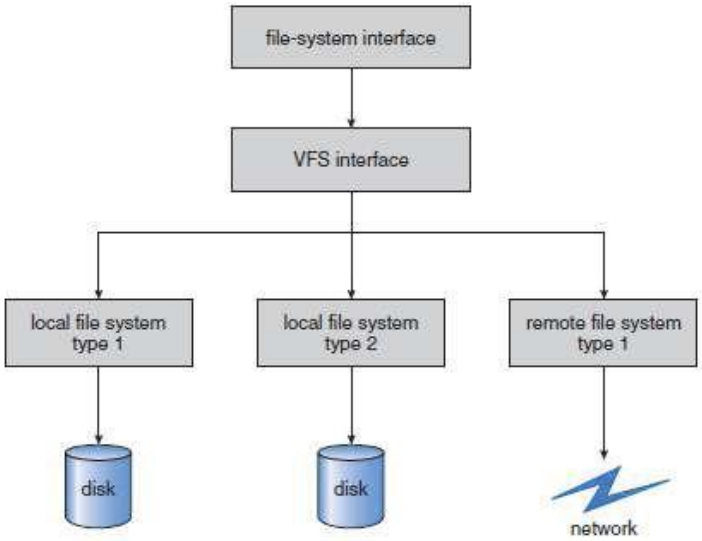
• Thus, the ***file-system implementation consists of three major layers***, as depicted schematically in Figure below.

- **The first layer** is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors.
- **The second layer** is called the virtual file system (VFS) layer; it serves two important functions:
    1. It separates file-system-generic operations from their implementation by defining a clean VFS interface
    2. The VFS provides a mechanism for uniquely representing a file throughout a network.

    The VFS is based on a file-representation structure, called a vnode, that contains a numerical designator for a network-wide unique file.

-Thus, the **VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.**

-**The third layer** of the architecture is the layer implementing the file system type or the remote-file-system protocol



5.9 Directory Implementation

• The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system.

• Algorithms for implementing Directory
    1) Linear-list
    2) Hash-table

5.9.1 Linear List

• A linear-list of file-names has pointers to the data-blocks.

• To create a new file:
    1) First search the directory to be sure that no existing file has the same name.
    2) Then, add a new entry at the end of the directory.

• To delete a file:
    1) Search the directory for the named-file and
    2) Then release the space allocated to the file.

• To reuse the directory-entry, there are 3 solutions:

1) Mark the entry as unused (by assigning it a special name).

2) Attach the entry to a list of free directory entries.

3) Copy the last entry in the directory into the freed location & to decrease length of directory.

• **Problem:** Finding a file requires a linear-search which is slow to execute**.**

Solutions:

1) A cache can be used to store the most recently used directory information.

2) A **sorted list** allows a binary search and decreases search time.

• Advantage: Simple to program.

• Disadvantage:Time-consuming to execute.

Solution : A more sophisticated tree data structure, such as a B-tree, might help here

5.9.2 Hash Table

• A linear-list stores the directory-entries.

• In addition, a hash data-structure is also used.

• The hash-table

→ takes a value computed from the file name and

→ returns a pointer to the file name in the linear-list.

• Advantages:

1) Decrease the directory search-time.

2) Insertion & deletion are easy.

• Disadvantages:

1**) collisions** i.e. a situation in which 2 file-names hash to the same location.

2) Fixed size of hash-table and the dependence of the hash function on that size.

• To enlarge **linear-probing hash table** we must reorganize the existing directory entries to reflect their new hash-function values.

Solution: a **chained-overflow hash table** can be used.

-Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

-Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries.

-Still, this method is much faster than a linear search.

5.10 Allocation Methods

• The direct-access nature of disks allows us flexibility in the implementation of files.

• In almost every case, many files are stored on the same disk.

• Main problem: How to allocate space to the files so that

→ disk-space is utilized effectively and

→ files can be accessed quickly.

• Three methods of allocating disk-space:

1) Contiguous

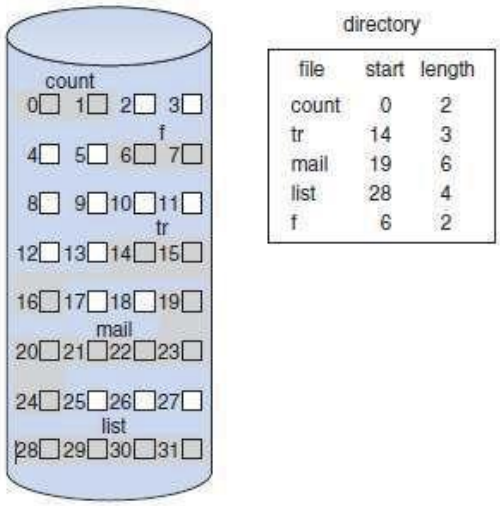
2) Linked and

3) Indexed

• Each method has advantages and disadvantages.

5.10.1 Contiguous Allocation

• Each file occupies a set of contiguous-blocks on the disk (Figure 4.30).
• Disk addresses define a linear ordering on the disk.
• The number of disk seeks required for accessing contiguously allocated files is minimal.
• Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.
• If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + n — 1.
• The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file this is illustrated in the figure .
• Both **sequential and direct access can be supported**.



• Problems:

1) **Finding space for a new file**

- External fragmentation can occur.

2) **Determining how much space is needed for a file**.

- If you allocate too little space, it can't be extended.

Two solutions:

i) The user-program can be terminated with an appropriate error-message. The user must then allocate more space and run the program again.

ii) Find a larger hole, copy the contents of the file to the new space and release the previous space.

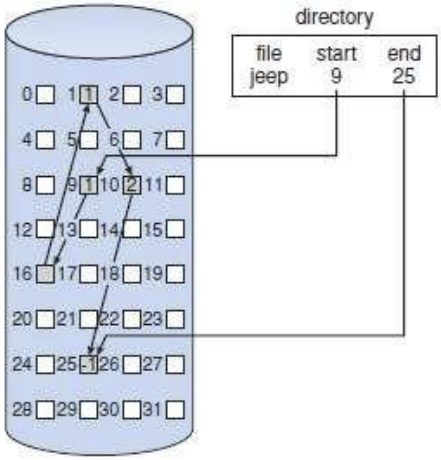
• To minimize these drawbacks:

1) A contiguous chunk of space can be allocated initially and

2) Then when that amount is not large enough, another chunk of contiguous space known as an **extent is added**.

-The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

5.10.2 Linked Allocation



• Each file is a linked-list of disk-blocks.
• The disk-blocks may be scattered anywhere on the disk.
• The directory contains a pointer to the first and last blocks of the file
• For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (as shown in Figure).
E• ach block contains a pointer to the next block.
• These pointers are not made available to the user.
• To create a new file,
- a new entry in the directory is created.
-each directory entry has a pointer to the first disk block of the file.
-This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file.
-The size field is also set to 0.
-**A write** to the file causes a free block to be found. This new block is then written to and linked to the eof (end of file).
- **A read** to the file causes moving the pointers from block to block.

• Advantages:

1**) No external fragmentation, and any** free block on the free-space list can be used to satisfy a request.

2) **The size of the file doesn't need to be declared on creation**.

3) Not necessary to **compact disk-space.**

• Disadvantages:

1) Can be used effectively **only for sequential-access files**.

-To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.

- Each access to a pointer requires a disk read, and some require a disk seek.

2) Space required for the pointers.

Solution:

-Collect blocks into multiples called clusters & allocate clusters rather than blocks.

-Pointers then use a much smaller percentage of the file's disk space.

• The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.

2) **Reliability**: Problem occurs if a pointer is lost(or damaged).

Partial solutions: i) Use doubly linked-lists.

ii) Store file name and relative block-number in each block.

-however, these schemes require even more overhead for each file.

File Allocation Table

• FAT is a variation on linked allocation (FAT=File Allocation Table).

• A section of disk at the beginning of each partition is set aside to contain the table.

• **The table**

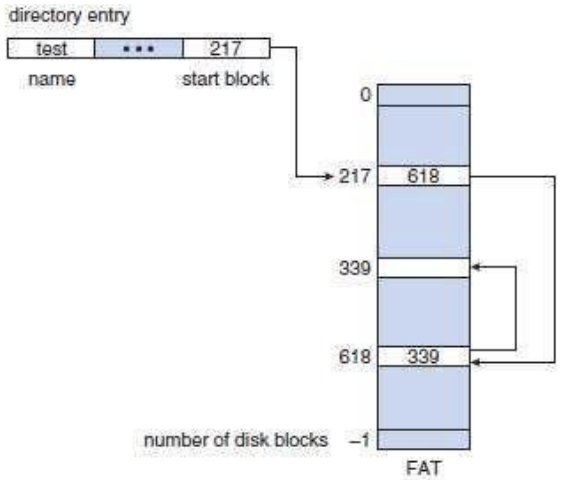
→ has one entry for each disk-block and

→ is indexed by block-number.

• The directory-entry contains the block-number of the first block in the file.

• The table entry indexed by that block-number then contains the block-number of the next block in the file.

• This chain continues until the last block, which has a special end-of-file value as the table entry.



• Unused blocks are indicated by a 0 table value.

• Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block.

• The 0 is then replaced with the end-of-file value.

An illustrative example is the FAT structure shown in Figure for a file consisting of disk blocks 217, 618, and 339.

• Advantages:

1) FAT Cache can be used to reduce the no. of disk head seeks.

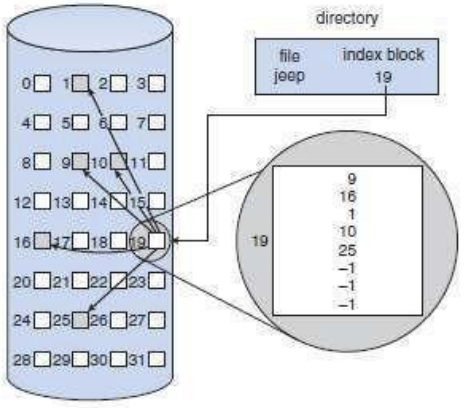
2) Improved random access time, since the disk head can find the location of any block by reading the information in the FAT.

5.10.3 Indexed Allocation

• Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.

• However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.

Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.



• Each file has its own index block, which is an array of disk-block addresses.

• The ith entry in the index block points to the ith file block (as shown in Figure).

• The directory contains the address of the index block.

• Advantage:

1) Supports direct access, without external fragmentation,because any free block on the disk can satisfy a request for more space.

• Disadvantages:

1) Suffer from wasted space,

-an entire index block must be allocated, even if only one or two pointers will be non-nil.

2) The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation

-If the index block is too small, however, it will not be able to hold enough pointers for a large file, and to deal with this issue we have the following mechanisms:

1) Linked Scheme

- To allow for large files, link several index blocks,

2) Multilevel Index

- A first-level index block points to second-level ones, which in turn point to the file blocks,

3) Combined Scheme

-Used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode.

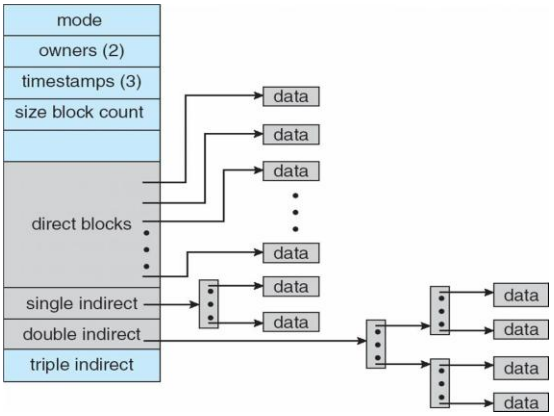
-The **first 12** of these pointers point to direct blocks; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block.

-The **next three pointers** point to indirect blocks.

-**The first pointer** points to a single indirect block, which is an index block containing not data but the addresses of blocks that do
contain data.

-**The second pointer** points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.

-**The last pointer** contains the address of a triple indirect block.



-A UNIX inode which uses Combined Scheme is shown in Figure .

5.10.4 Performance

• Contiguous Allocation

-Requires only one access to get a disk-block

-We can calculate immediately the disk address of the next block and read it directly

-Good for direct access

• Linked Allocation

-Good for sequential access

-Not be used for an application requiring direct access

• Indexed Allocation

- If the index block is already in memory, then the access can be made directly

- keeping the index block in memory requires considerable space

5.11 Free Space Management

• A free-space list keeps track of free disk-space (i.e. those not allocated to some file or directory).

• To create a file,

1) We search the free-space list for the required amount of space.

2) Allocate that space to the new file.

3) This space is then removed from the free-space list.

• when a file is deleted , its disk-space is added to the free-space list.

1) Bit Vector

• The free-space list is implemented as a bit map/bit vector.

• Each block is represented by a bit.

1) If the block is free, the bit is 1.

2) If the block is allocated, the bit is 0.

• For example, consider a disk where blocks 2, 3, 4, 5 and 7 are free and the rest of the blocks are allocated. The free-space bit map will be 00111101

• Advantage:

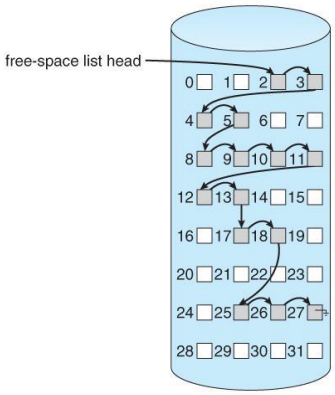
1) Relative simplicity & efficiency in finding the first free block, n consecutive free blocks.

• Disadvantages:

1) Inefficient unless the entire vector is kept in main memory.

2) The entire vector is written to disc occasionally for recovery.

2) Linked List



• The basic idea:

1) Link together all the free disk-blocks

2) Keep a pointer to the first free block in a special location on the disk.

3) Cache the block in memory.

• The first block contains a pointer to the next free one, etc.

• For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25,26, and 27 are free and the rest of the blocks are allocated.

•A pointer to block 2 is kept as the first free block.

Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 7, and so on (as shown in figure).

• Disadvantage:

1) Not efficient, because to traverse the list, each block is read,which consumes more I/O time.

• Usually the OS simply needs a free block, and uses the first one.

• The FAT method incorporates free-block accounting into the allocation data structure. No separate method is needed

3) Grouping

• The addresses of n free blocks are stored in the 1st free block.

• The first n-1 of these blocks are actually free.

• The last block contains addresses of another n free blocks, etc.

• Advantage:

1) Addresses of a large no of free blocks can be found quickly.

4) Counting

• Takes advantage of the fact that, generally, several contiguous blocks may be allocated/freed simultaneously.

• Keep the address of the first free block and the number n, of free contiguous blocks that follow the first block.

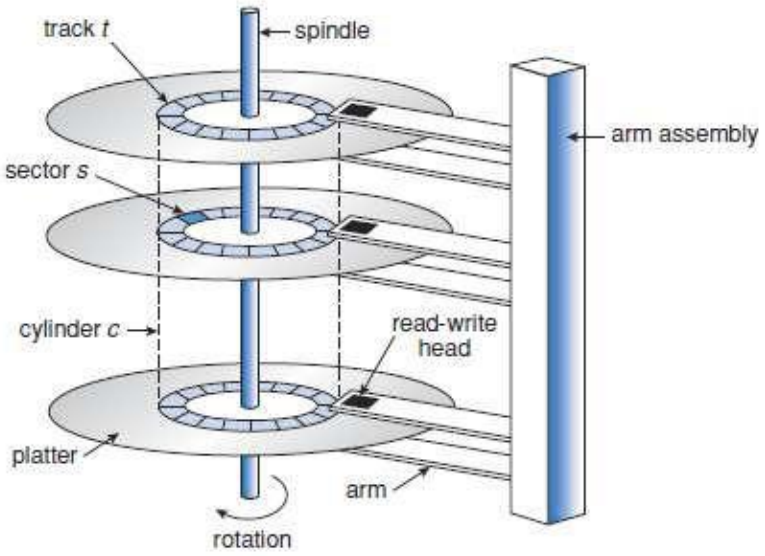
• Each entry in the free-space list then consists of a disk address and a count.

MASS-STORAGE STRUCTURE

5.12 Mass Storage Structures

5.12.1 Magnetic Disks

• Hard-disks provide the **bulk of secondary-storage** for modern computer-systems
• Each disk-platter has a flat circular-shape, like a CD.
• The 2 surfaces of a platter are covered with a magnetic material.
• Information is stored on the platters by recording magnetically.



• A read-write head flies just above the surface of the platter.
• The heads are attached to a disk-arm that moves all the heads as a unit.
• The surface of a platter is logically divided into circular tracks, which are subdivided into sectors.
• The set of tracks that are at one arm position makes up a cylinder.
• There may be thousands of concentric-cylinders in a disk-drive, and each track may contain hundreds of sectors.
• Disk-speed has 2 parts:

1) The transfer-rate is the rate at which data flow between the drive and the computer.
2) The positioning-time(or random-access time) consists of 2 parts:
   i) Seek-time refers to the time necessary to move the disk-arm to the desired cylinder.
   ii) Rotational-latency refers to the time necessary for the desired sector to rotate to the disk-head.

•Although the disk platters are coated with a thin protective layer, sometimes the head will damage the magnetic surface. This accident is called a **head crash.** A head crash normally cannot be repaired; the entire disk must be replaced.
• A disk can be removable which allows different disks to be mounted as needed.
• A disk-drive is attached to a computer by an I/O bus.

• Different kinds of buses:

→enhanced integrated drive electronics (EIDE),
→ advanced technology attachment (ATA) → serial ATA (SATA)
→ universal serial bus (USB)
→ fibre channel (FC) and SCSI(small computer interface)

• Data transfers

-The data transfers on a bus are carried out by special electronic processors called **controllers.**

-The **host controller** is the controller at the computer end of the bus.

-A **disk controller** is built into each disk drive.

-To perform a disk I/O operation, the computer places a command into the host controller,

-The host controller then sends the command via messages to the disk controller, and

-The disk controller operates the disk-drive hardware to carry out the command.

-Disk controllers usually have a built-in cache.

-Data transfer at the disk drive happens between the cache and the disk surface, and

-Data transfer to the host occurs between the cache and the host controller.

5.12.2 Magnetic Tapes

• Magnetic tape was used as an early secondary-storage medium.

• Advantages:

-It is relatively permanent and can hold large quantities of data.

• Disadvantages:

1) Its access time is slow compared with that of main memory and Hard-disk.

2) In addition, random access to magnetic tape is about a thousand times slower than random access to Magnetic disk, so tapes are not very useful for secondary-storage.

• Applications:

1) Tapes are used mainly for backup, for storage of infrequently used information.

2) Tapes are used as a medium for transferring information from one system to another.

• A tape is kept in a spool and is wound or rewound past a read-write head.

• Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives.

5.13 Disk Structure

• Modern Hard-disk-drives are addressed as large one-dimensional arrays of logical blocks.

• The logical block is the smallest unit of transfer.

• The size of a logical block is usually 512 bytes

• One-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially:

-Sector 0 is the first sector of the first track on the outermost cylinder.

-The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

• In practice, it is difficult to perform this mapping, for two reasons.

1) Most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk.

2) The number of sectors per track is not a constant on some drives.

➢ **Constant linear velocity (CLV),**

-The density of bits per track is uniform.

-The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold.

-As we move from outer zones to inner zones, the number of sectors per track decreases.

-Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone.

-The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head.
-This method is used in CD-ROM and DVD-ROM drives.

- **constant angular velocity (CAV).**

-the disk rotation speed can stay constant, and
-the density of bits decreases from inner tracks to outer tracks to keep the data rate constant.
-This method is used in hard disks and is known as **constant** angular **velocity (CAV).**

5.14 Disk Attachment

• Computers access disk storage in two ways.

1) Via I/O ports (or host-attached storage); this is common on small systems.
2) Via a remote host in a distributed file system; this is referred to as network-attached storage.

5.14.1 Host-Attached Storage

• Host-attached storage is **storage accessed through local I/O ports.**
• These ports use several technologies.

1) **The desktop PC** uses an **I/O bus architecture called IDE or ATA.**
-This architecture supports a **maximum of 2 drives per I/O bus**.
2) **High-end workstations and servers** generally use more sophisticated I/O architectures, such as SCSI and fiber channel (FC).

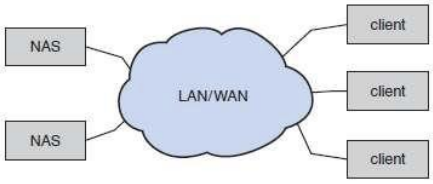
• **Fiber channel (FC**), a high-speed serial architecture that can operate over optical fiber.
-It has two variants:
i) One is a large switched fabric having a 24-bit address space.
-This variant is the basis of storage-area networks (SANs).
ii) The other FC variant is an arbitrated loop (FC-AL) that can address 126 devices.

• **SCSI** is a bus parallel architecture.
- Its physical medium is usually **a ribbon cable** having a large number of conductors .
- The SCSI protocol supports a **maximum of 16 devices** on the bus.
- The devices include one controller card in the host (the **SCSI initiator)** and up to **15 storage devices**(the **SCSI targets).**

• A wide variety of storage devices are suitable for use as host-attached storage.
For ex: Hard-disk-drives, RAID arrays, and CD, DVD, and tape drives.

5.14.2 Network-Attached Storage

• A network-attached storage (NAS) device is a special-purpose storage system that is **accessed remotely over a data network (as shown in Figure).**



• Clients access NAS via a **remote-procedure-call interface** such as
→ NFS for UNIX systems
→ CIFS for Windows machines.

• The remote procedure calls (RPCs) are carried via TCP or UDP over a local area network (LAN).

• Usually, the **same local area network (LAN)** carries all data traffic to the clients.

• The **NAS device is usually implemented as a RAID array** with software that implements the RPC interface.

[A variety of disk-organization techniques, collectively called redundant arrays of inexpensive disks (RAIDS), are commonly used to address the performance and reliability issues.]

• Advantage:
-All computers on a LAN can **share a pool of storage** with the same ease of naming and access with local host-attached storage.

• Disadvantages:
1) **NAS is less efficient and have lower performance** than some direct-attached storage options.
2) The **storage I/O operations consume bandwidth on the data network**, thereby increasing the latency of network communication.

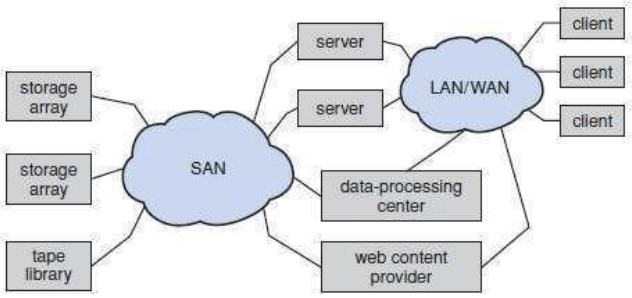
• iSCSI(internet SCSI) is the latest network-attached storage protocol.

• iSCSI uses the IP network protocol to carry the SCSI protocol.

5.14.3 Storage-Area Network

• A storage-area network (SAN) is a private network (using **storage protocols** rather than networking protocols) connecting servers and storage units, as shown in Figure .

• The power of a SAN lies in its flexibility.
1) Multiple hosts and multiple storage-arrays can attach to the same SAN.
2) Storage can be dynamically allocated to hosts.
3) A SAN switch allows or prohibits access between the hosts and the storage.
4) SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections.
5) SANs typically have more ports and less expensive ports, than storage-arrays.



• **FC** is the most common **SAN interconnect**.

• Another SAN interconnect is **InfiniBand** — a special-purpose bus architecture that provides **hardware and software** support **for high-speed interconnection** networks for servers and storage units.

5.15 Disk Scheduling

• One of the responsibilities of the operating system is to use the **hardware efficiently**.

• For the disk drives, to meet this responsibility, it requires to **have fast access time and large disk bandwidth**.

• Access time has two major components:

1**) Seek-time**: The seek-time is the time for the disk-arm to move the heads to the cylinder containing the desired sector.

2) **Rotational-latency**: The Rotational-latency is the additional time for the disk to rotate the desired sector to the disk-head.

• The **disk bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

• **We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced**.

• Whenever a process needs I/0 to or from the disk, it issues a system call to the operating system.

• The request specifies several pieces of information:

1) Whether this operation is input or output

2) What the disk address for the transfer is

3) What the memory address for the transfer is

4) What the number of sectors to be transferred is

• If the **desired disk-drive and controller are available**, the request can be **serviced immediately**.

• If the **drive or controller is busy**, any new requests for service will be placed in the **queue** of pending requests for that drive.

• For a multiprogramming system with many processes, the disk queue may often have several pending requests.

• Thus, when one request is completed, the operating system chooses which pending request to service next.

• Any one of several disk-scheduling algorithms can be used.

5.15.1 FCFS Scheduling

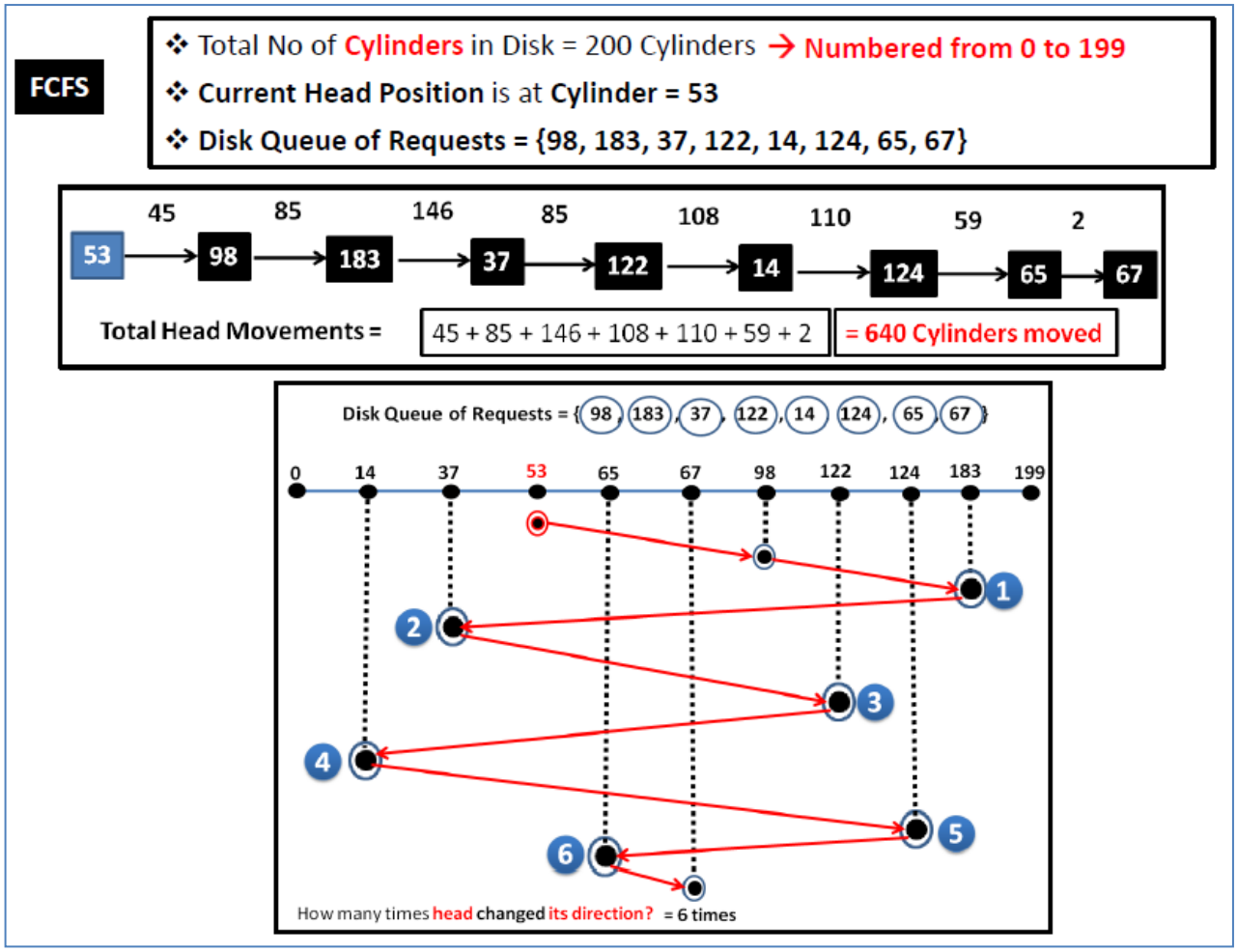
• FCFS stands for First Come First Serve.
• The requests are serviced in the same order, as they are received.
• Advantage: This algorithm is simple & fair.
• Disadvantage: Generally, this algorithm does not provide the fastest service.

Example:

Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122, 14, 124, 65, 67, in that order. The disk head is initially at cylinder 53.

FCFS:

Starting with cylinder 53, the disk-head will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67 as shown in Figure below.



5.15.2 SSTF Scheduling

• SSTF stands for Shortest Seek-time First.
• This selects the request with minimum seek-time from the current head-position.
• Problem: Seek-time increases with the number of cylinders traversed by head.

Solution: To overcome this problem, SSTF chooses the pending request closest to the current head-position.

• Advantage: SSTF is a substantial improvement over FCFS, it is not optimal.
• Disadvantage: Essentially, SSTF is a form of SJF and it may cause starvation of some requests

Example:

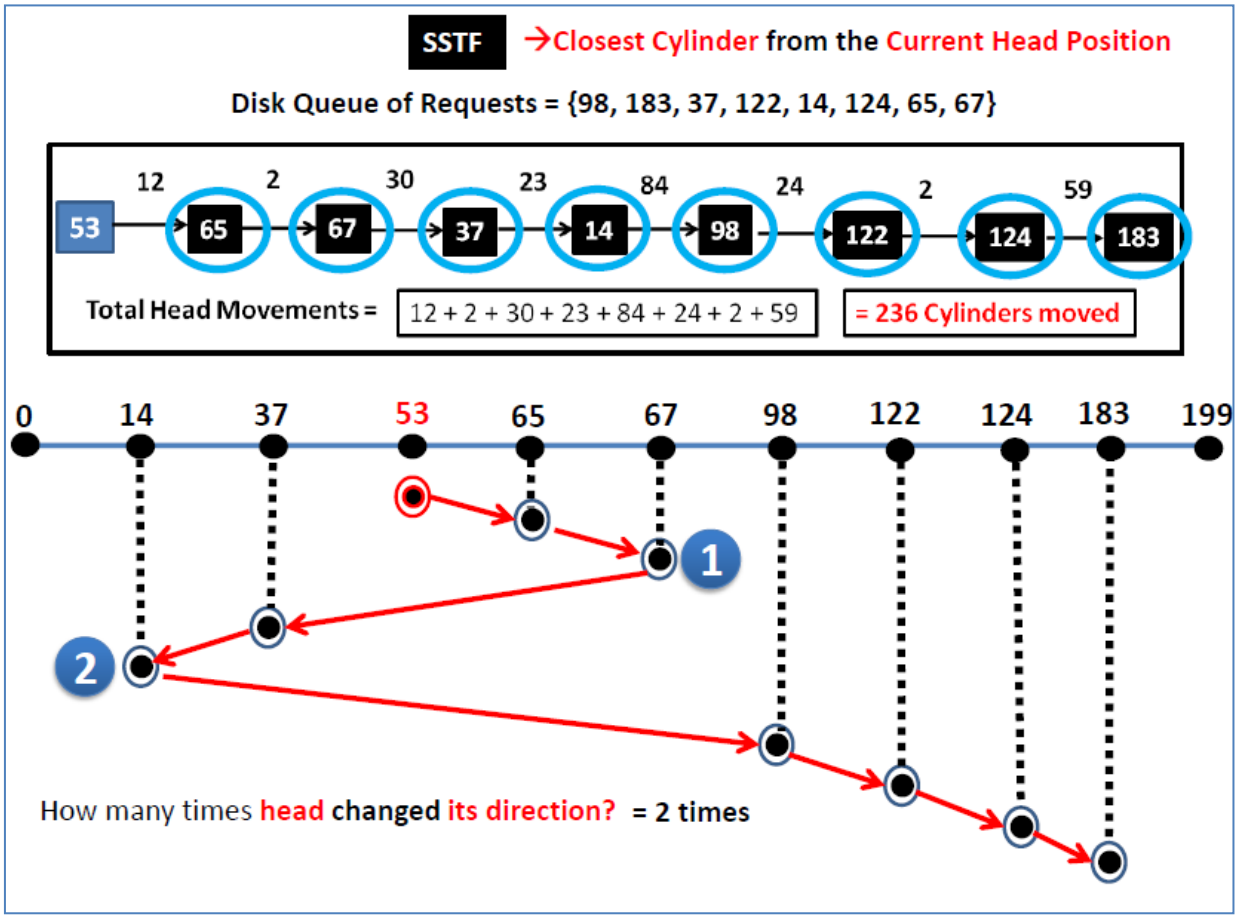
Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122, 14, 124, 65, 67, in that order. The disk head is initially at cylinder 53.

SSTF:

• The closest request to the initial head position 53 is at cylinder 65.

•Once we are at cylinder 65, the next closest request is at cylinder 67.

• From there, the request at cylinder 37 is closer than 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183. It is shown in Figure below.



5.15.3 SCAN Scheduling (elevator)

• The SCAN algorithm working:

1. The disk-arm starts at one end of the disk.
2. Then, the disk-arm moves towards the other end, servicing the request as it reaches each cylinder.
3. At the other end, the direction of the head movement is reversed and servicing continues.

• The head continuously scans back and forth across the disk.

• Thus the SCAN algorithm is sometimes called the elevator algorithm, since the disk-arm behaves just like an elevator in a building.

• Before applying SCAN algorithm, we need to know the current direction of head movement.

• **Disadvantage:** If a request arrives just in from of head, it will be serviced immediately.

-On the other hand, if a request arrives just behind the head, it will have to wait until the arms reach other end and reverses direction.

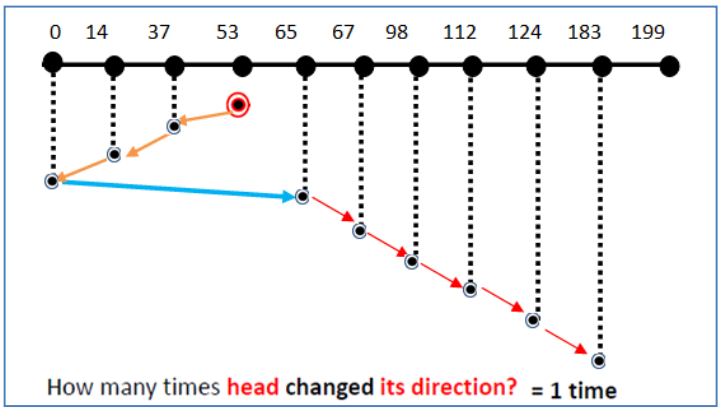
• Example:

Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122, 14, 124, 65, 67, in that order. The disk head is initially at cylinder 53. Assume that disk-arm is moving toward 0.

SCAN:

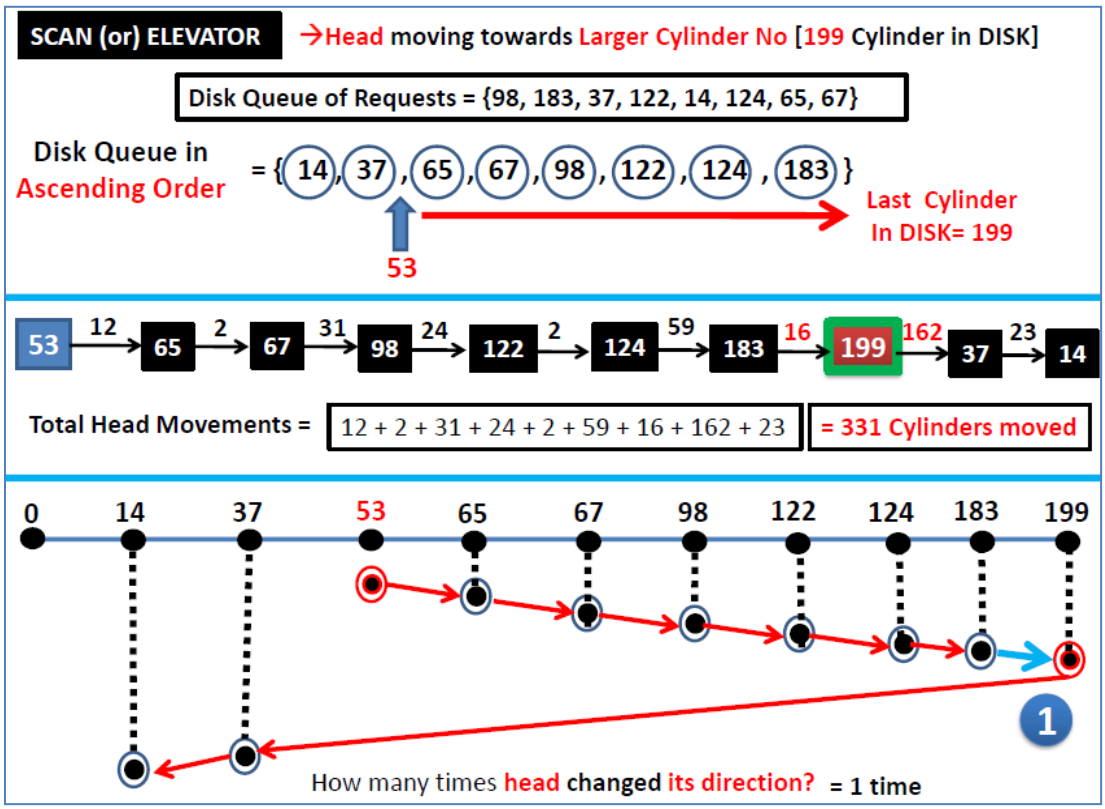
• Since arm is moving towards 0, the head will service first 37 and then 14.

• At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65,67,98, 122, 124, and 183. It is shown in Figure below.



Total head movement = 236

Example2: same problem with assumption that head(disk arm) moving towards Larger Cylinder No [199 Cylinder in DISK]


SCAN (or) ELEVATOR
→Head moving towards Larger Cylinder No [199 Cylinder in DISK]
Disk Queue of Requests = {98, 183, 37, 122, 14, 124, 65, 67}
Disk Queue in Ascending Order
= {14, 37, 65, 67, 98, 122, 124, 183}
53
Last Cylinder In DISK= 199
53 12 65 2 67 31 98 24 122 2 124 59 183 16 199 162 37 23 14
Total Head Movements = 12 + 2 + 31 + 24 + 2 + 59 + 16 + 162 + 23
= 331 Cylinders moved
0 14 37 53 65 67 98 122 124 183 199
1
How many times head changed its direction? = 1 time

5.15.4 C-SCAN Scheduling

• Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

• Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.

• When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip

• The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

• Before applying C - SCAN algorithm, we need to know the current direction of head movement.

• Example:

Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122, 14, 124, 65, 67, in that order. The disk head is initially at cylinder 53. Assume that disk-arm is moving towards Larger Cylinder No i.e,199

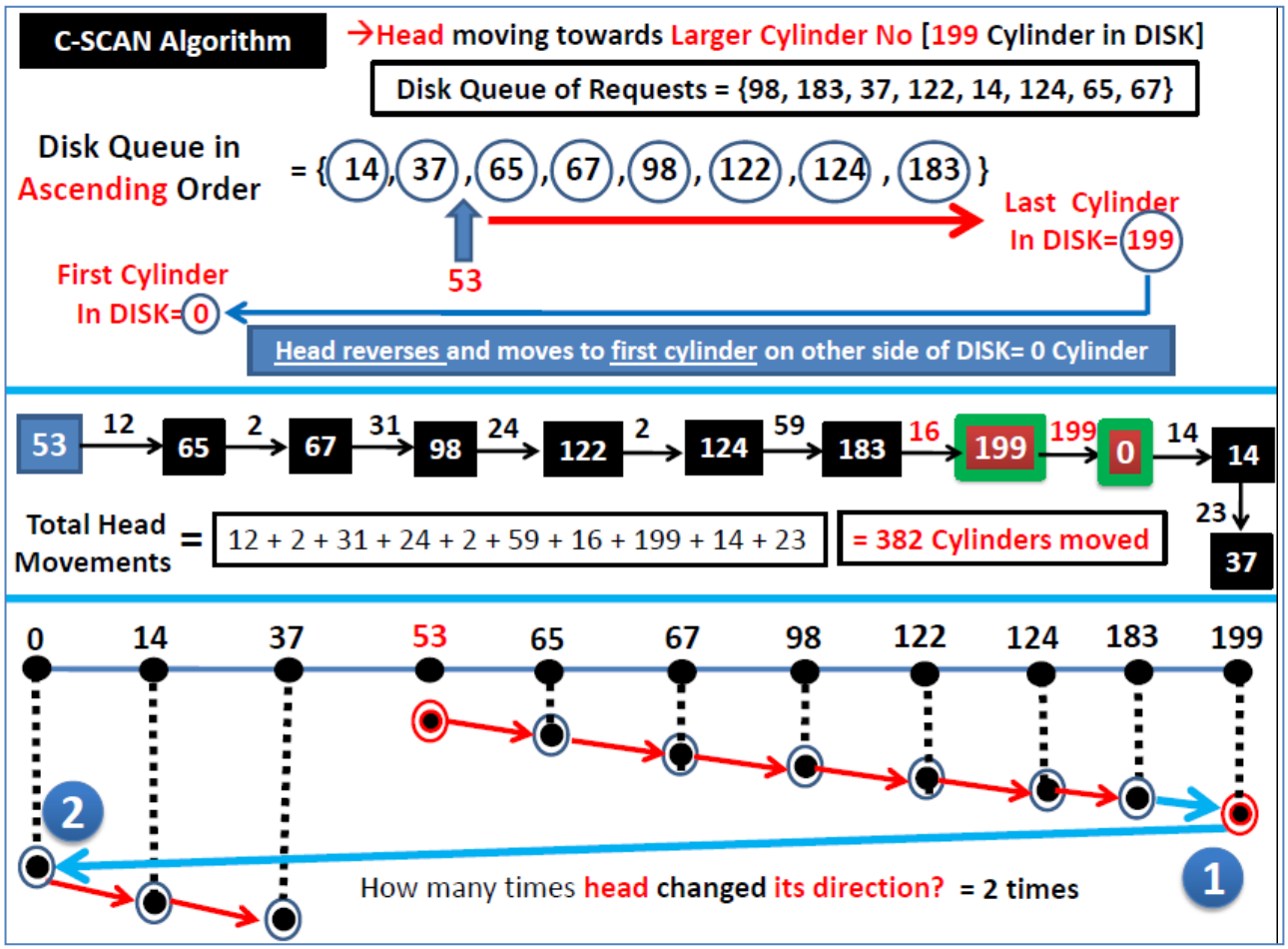
C-SCAN:

• Since arm is moving towards 199, the head will service 65, 67, 98, 122, 124, 183.

• Then it will move to 199 and the arm will reverse and move towards 0.

• While moving towards 0, it will not serve.

• But, after reaching 0, it will reverse again and then serve 14 and 37. It is shown in Figure below

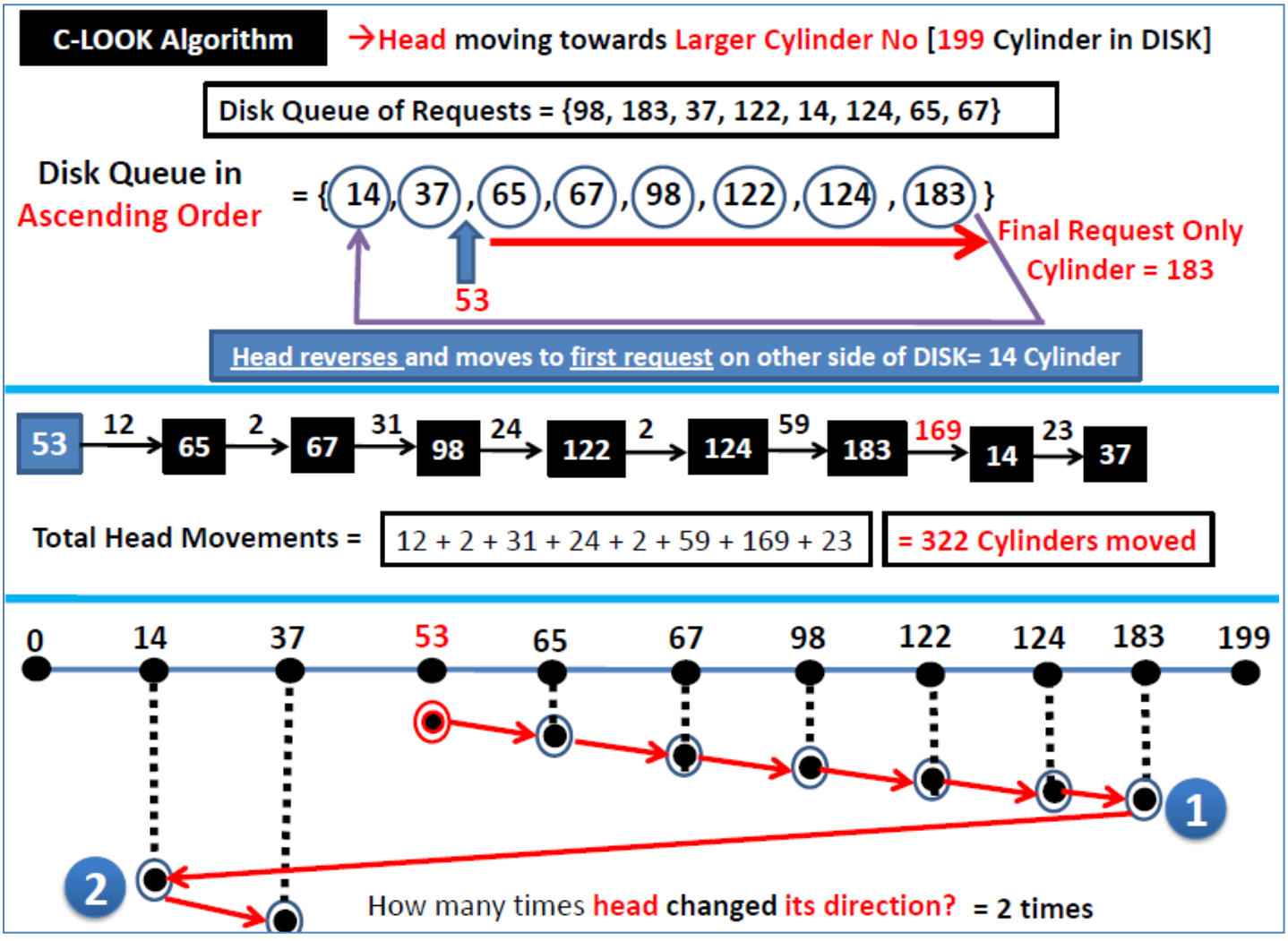

C-SCAN Algorithm
→Head moving towards Larger Cylinder No [199 Cylinder in DISK]
Disk Queue of Requests = {98, 183, 37, 122, 14, 124, 65, 67}
Disk Queue in Ascending Order
= { 14 , 37 , 65 , 67 , 98 , 122 , 124 , 183 }
53
Last Cylinder In DISK= 199
First Cylinder In DISK= 0
Head reverses and moves to first cylinder on other side of DISK= 0 Cylinder
53 12 65 2 67 31 98 24 122 2 124 59 183 16 199 199 0 14 14 23 37
Total Head Movements = 12 + 2 + 31 + 24 + 2 + 59 + 16 + 199 + 14 + 23
= 382 Cylinders moved
0 14 37 53 65 67 98 122 124 183 199
2
1
How many times head changed its direction? = 2 times

5.15.5 LOOK Scheduling

• SCAN and C-SCAN algorithm move the disk-arm across the full width of the disk.
• In practice, the SCAN algorithm is not implemented in this way.
• The arm goes only as far as the final request in each direction.
• Then, the arm reverses, without going all the way to the end of the disk.
• This version of SCAN and C-SCAN is called LOOK and C-LOOK scheduling because they look for a request before continuing to move in a given direction.
• Before applying algorithm, we need to know the current direction of head movement
• **Example:**

Consider, for example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37,122, 14, 124, 65, 67, in that order. The disk head is initially at cylinder 53. Assume that disk-arm is moving towards Larger Cylinder No i.e,199

• Since arm is moving towards 199, the head will service 65, 67, 98, 122, 124, 183
• Then the arm will reverse and move towards 14. Then it will serve 37. It is shown in Figure below.


C-LOOK Algorithm
→Head moving towards Larger Cylinder No [199 Cylinder in DISK]
Disk Queue of Requests = {98, 183, 37, 122, 14, 124, 65, 67}
Disk Queue in Ascending Order
= {14, 37, 65, 67, 98, 122, 124, 183}
53
Final Request Only Cylinder = 183
Head reverses and moves to first request on other side of DISK= 14 Cylinder
53 12 65 2 67 31 98 24 122 2 124 59 183 169 14 23 37
Total Head Movements = 12 + 2 + 31 + 24 + 2 + 59 + 169 + 23
= 322 Cylinders moved
0 14 37 53 65 67 98 122 124 183 199
1
2
How many times head changed its direction? = 2 times

5.15.6 Selection of a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal because it increases performance over FCFS.
- SCAM and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem.
- For any particular list of requests, we can define an optimal order of retrieval, but the computation needed to find an optimal schedule may not justify the savings over SSTF or SCAN.
- With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request.
- Then, all scheduling algorithms behave the same, because they have only one choice for where to
- move the disk head: They all behave like FCFS scheduling.

➢ Requests for disk service can be greatly influenced by the file-allocation method.

- A program reading a contiguously allocated file will generate several requests that are close together on the disk, resulting in limited head movement.
- A linked or indexed file, in contrast, may include blocks that are widely scattered on the disk, resulting in greater head movement.

➢ The location of directories and index blocks is also important.

- Since every file must be opened to be used, and opening a file requires searching the directory structure, the directories will be accessed frequently.
- Suppose that a directory entry is on the first cylinder and a file's data are on the final cylinder.
- In this case, the disk head has to move the entire width of the disk.
- If the directory entry were on the middle cylinder, the head would have to move, at most, one-half the width. Caching the directories and index blocks in main memory can also help to reduce the disk-arm movement, particularly for read requests.

➢ Because of these complexities, the disk-scheduling algorithm should be written as a separate module of the operating system, so that it can be replaced with a different algorithm if necessary.

Either SSTF or LOOK is a reasonable choice for the default algorithm.

5.16 Disk Management

• The operating system is responsible for several other aspects of disk management.
• For example:
1) disk initialization
2) booting from disk
3) bad-block recovery.

5.16.1 Disk Formatting

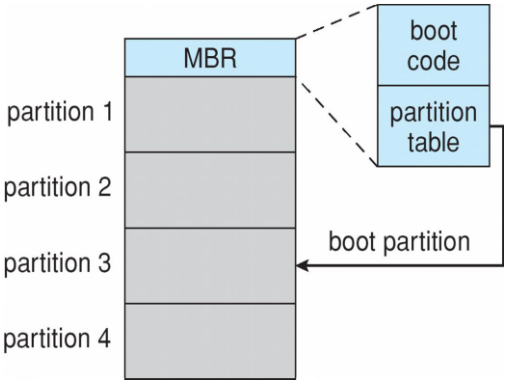
• Usually, **a new Hard-disk is a blank slate**: it is just a platter of a magnetic recording material.
• Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called **low-level formatting**, or physical formatting.
• Low-level formatting fills the disk with a special data structure for each sector.
• The data structure for a sector typically consists of
→ a header
→ a data area (usually 512 bytes in size), and
→ a trailer.
• The header and trailer contain information used by the disk controller, such as
→ sector number and
→ error-correcting code (ECC).
-When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area.
-When the sector is read, the ECC is recalculated and is compared with the stored value.
-If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad.
-The ECC is an *error-correcting* code because it contains enough information that, if only a few bits or data have been corrupted, the controller can identify which bits, have changed and can calculate what their correct values should be.
-It then reports a recoverable soft error.
-The controller automatically does the ECC processing whenever a sector is read or written
• Before a disk can store data, the operating system still needs to record its own data structures on the disk.
• It does so in two steps.
1) Partition the disk into one or more groups of cylinders.
-The operating system can treat each partition as a separate disk.
- For example:
one partition can hold a copy of the operating system's executable code, another partition can hold user files.
2) Logical formatting, or creation of a file system.
- The operating system stores the initial file-system data structures onto the disk.
- These data structures may include maps of free and allocated space and an initial empty directory.
• To increase efficiency, most file systems group blocks together into larger chunks, frequently called clusters. 1) Disk I/O is done via blocks,
2) File system I/O is done via clusters.

5.16.2 Boot Block

• For a computer to start running, it must have a bootstrap program to run.
• Bootstrap program
→ initializes CPU registers, device controllers and the contents of main memory and
→ then starts the operating system.
• For most computers, the bootstrap is stored in read-only memory (ROM).
• Main Problem: To change the bootstrap code, the ROM hardware chips has to be changed.
• To solve this problem, most systems store a tiny bootstrap loader program in the boot-ROM.
• Job of boot-ROM: Bring in a full bootstrap program from disk.
• The full bootstrap program can be changed easily: ―A new version is simply written onto the disk‖.
• The full bootstrap program is stored in the ―boot blocks at a fixed location on the disk.
• A disk that has a boot partition is called a boot disk or system disk.
• In the boot-ROM, the code
→ instructs the disk-controller to read the boot blocks into memory and
→ then starts executing that code.

➢ **Example: the boot process in Windows 2000.**

• The Windows 2000 system places its boot code in the first sector on the hard disk (which it terms the **master boot record,** or MBR).
• a hard diskis allowed to be divided into one or more partitions;
• one partition, identified as the **boot partition,** contains the operating system and device drivers.
• Booting begins in a Windows 2000 system by running code that is resident in the system's ROM memory.
• This code directs the system to read the boot code from, the MBR.
• In addition to containing boot code, the MBR contains a table listing the partitions for the hard disk and a flag indicating which partition the system is to be booted from.
• This is illustrated in Figure.



• Once the system identifies the boot partition, it reads the first sector from that partition (which is called the **boot sector)** and continues with the remainder of the boot process, which includes loading the various subsystems and system services..

5.16.3 Bad Blocks

• Because disks have moving parts and small tolerances, they are prone to failure.
• Sometimes,
→ The disk needs to be replaced.
→ The disk-contents need to be restored from backup media to the new disk.
→ One or more sectors may become defective.
• Most disks even come from the factory with bad blocks

• How to handle bad-blocks?

- **handled manually**
  - On simple disks, bad-blocks are handled manually.
  - One strategy is to scan the disk to find bad-blocks while the disk is being formatted.
  - Any bad-blocks that are discovered are flagged as unusable in FAT entry.
  -Thus, the file system does not allocate them.
  - If blocks go bad during normal operation, a special program must be run manually
    → to search for the bad-blocks and
    → to lock the bad-blocks.
  - Usually, data that resided on the bad-blocks are lost.

- **sector sparing** or **forwarding**
  -The controller maintains a list of bad blocks on the disk.
  -The list is initialized during the low-level formatting at the factory and is updated over the life of the disk.
  -Low-level formatting also sets aside spare sectors not visible to the operating system.
  -The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding.**

• A typical bad-sector transaction might be as follows:

1) The operating system tries to read logical block 87.

2) The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

3) The next time the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.

4) After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller.

- **sector slipping**
  - replace a bad block by sector slipping.

an example:

-Suppose that logical block 17 becomes defective and the first available spare follows sector 202.
-Then, sector slipping remaps all the sectors from 17 to 202, moving them all down one spot.
-That is, sector 202 is copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19.
-Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it.

5.17 Swap Space Management

• Swap-space management is a **low-level task of the operating system.**
• Virtual memory uses disk space as an extension of main memory.
• Main goal of swap space: to provide the best throughput for the virtual memory system.
• Here, we discuss about
  1) Swap space use
  2) Swap space location.

5.17.1 Swap-Space Use

• Swap space can be used in 2 ways.

1) Swapping-Systems may use swap space to hold an entire process image, including the code and data segments.

2) Paging-systems may simply store pages that have been pushed out of main memory.

• The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on

→ amount of physical memory,

→ amount of virtual memory it is backing, and

→ way in which the virtual memory is used.

5.17.2 Swap-Space Location

• A swap space can reside in one of two places:

1) The swap space can be a large file within **the file system.**

-Here, normal file-system routines can be used to create it, name it, and allocate its space.

- Advantage: This approach easy to implement,

- Disadvantage: This approach is inefficient. This is because

i) Navigating the directory structure and the disk structures takes time and extra disk accesses.

ii) External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image.

2) The swap space can be in a separate **raw (disk) partition**.

-No file system or directory structure is placed in the swap space.

-Rather, a separate swap-space storage manager is used to allocate and de-allocate the blocks from the raw partition.

-This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file system.

➢ An Example: Swap-Space Management in Linux

• The swap space is only **used for anonymous memory or for regions of memory shared by several processes**.

• Linux allows one or more swap areas to be established.

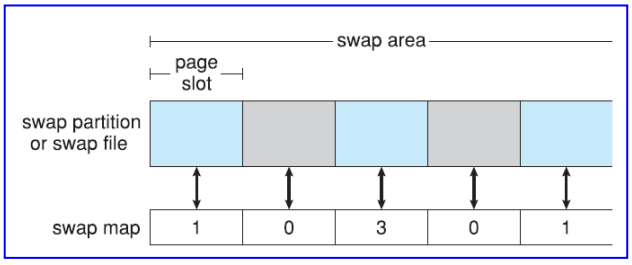
• A swap area may be in either a swap file on **a regular file system or a raw swap partition**.

• The data structures for swapping on Linux systems are shown in Figure

• Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages.

• Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area.

• If the value of a counter is 0, the corresponding page slot is available.



• Values greater than 0 indicate that the page slot is occupied by a swapped page.

• The value of the counter indicates the number of mappings to the swapped page;

• for example, a value of 3 indicates that the swapped page is mapped to three different processes (shared by three processes).

5.18 Protection vs. Security

Protection

• Protection controls access to the system-resources by
→ Programs
→ Processes or
→ Users.

• Protection ensures that only processes that have gained proper authorization from the OS can operate on
→memory-segments
→ CPU and
→ other resources.

• Protection must provide
→ means for specifying the controls to be imposed
→ means of enforcing the controls.

• Protection is an internal problem.

• Security, in contrast, must consider both the computer-system and the environment within which the system is used.

Security

• Security ensures the authentication of system-users to protect
→ integrity of the information stored in the system (both data and code)
→ physical resources of the computer-system.

• The security-system prevents
→ unauthorized access
→ malicious destruction
→ alteration of data or
→ accidental introduction of inconsistency.

5.19 Goals of Protection

• Operating system consists of a collection of objects, hardware or software.

• Each object has a unique name and can be accessed through a well-defined set of operations.

• Protection problem:
-ensure that each object is accessed correctly & only by those processes that are allowed to do so.

• Reasons for providing protection:
1) To prevent mischievous violation of an access restriction.
2) To ensure that each program component active in a system uses system resources only in ways consistent with policies.

• **Mechanisms are distinct from policies**:
1) Mechanisms determine how something will be done.
2) Policies decide what will be done.

• The separation of policy and mechanism is important for flexibility.

Policies are likely to change from place to place or time to time.

In the worst case, every change in policy would require a change in the underlying mechanism.

Using general mechanisms enables us to avoid such a situation.

5.20 Principles of Protection

• A key principle for protection is the principle of least privilege.

• **Principle of Least Privilege:**

- Programs, users, and even systems are given just enough privileges to perform their tasks‖.

• The principle of least privilege can help produce a more secure computing environment.

• An operating system which follows the principle of least privilege implements its features, programs, system-calls, and data structures so that failure of a component results in minimum damage.

• An operating system also provides system-calls and services that allow applications to be written with fine-grained access controls.

• Access Control provides mechanisms

→ to enable privileges when they are needed.

→ to disable privileges when they are not needed.

• **Audit-trails** for all privileged function-access can be created.

• Audit-trail can be used to trace all protection/security activities on the system.

• The audit-trail can be used by

→ Programmer

→ System administrator or

→ Law-enforcement officer.

• Managing users with the principle of least privilege requires creating a separate account for each user, with just the privileges that the user needs.

• Some systems implement role-based access control (RBAC) to provide this functionality.

• Computers implemented in a computing facility under the principle of least privilege can be limited to

→ running specific services

→ accessing specific remote hosts via specific services → accessing during specific times.

• Typically, these restrictions are implemented through enabling or disabling each service and through using Access Control Lists.

5.21 Domain of Protection

• A computer system is a collection of processes and objects.

• By *objects,* we mean both

-**hardware objects** (such as the CPU, memory segments, printers, disks, and tape drives) and

- software objects (such as files, programs, and semaphores).

• Each object has a unique name that differentiates it from all other objects in the system, and each can be accessed only through well-defined and meaningful operations.

• Objects are essentially abstract data types.

• The operations that are possible may depend on the object.

• For example, - a CPU can only be executed .

- Memory segments can be read and written,

- Data files can be created, opened, read, written, closed, and deleted and so on

• A process should be allowed to access only those resources for which it has authorization.

• at any time, a process should be able to access only those resources that it currently requires to complete its task.

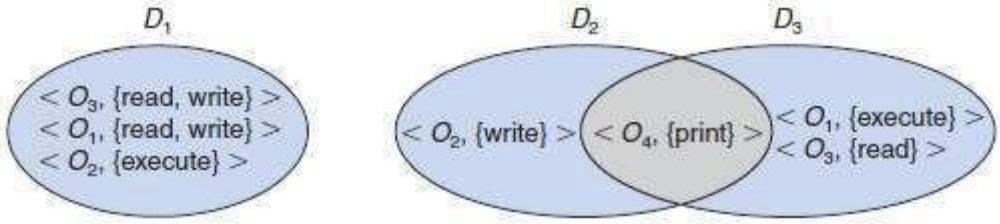
• This requirement, commonly referred to as the ***need-to-know*** **principle**, is useful in limiting the amount of damage a faulty process can cause in the system.

• For example, when process *p* invokes procedure *A()*, the procedure should be allowed to access only its own variables and the formal parameters passed to it; it should not be able to access all the variables of process *p*.

• The need-to-know principle is similar to the principle of least privilege that is the goals of protection are to minimize the risks of possible security violations.

5.21.1 Domain Structure

• A process operates within a protection domain.
• Protection domain specifies the resources that the process may access.
• Each domain defines
→ set of objects and
→ types of operations that may be invoked on each object.
• The ability to execute an operation on an object is an access-right.
• A domain is a collection of access-rights.
• The access-rights are an ordered pair <object-name, rights-set>.
• For example:
-If domain D has the access-right <file F, {read,write}>;
-Then a process executing in domain D can both read and write on file F.
• As shown in Figure below, domains may share access-rights.
The access-right <O4, {print}> is shared by D2 and D3.



• The association between a process and a domain may be either static or dynamic.

1) If the association between processes and domains is **static means** the set of resources available to the process is **fixed throughout** the process's lifetime.

- then a mechanism must be available to change the content of a domain ,this is required to always reflects the minimum necessary access rights.

2) If the association between processes and domains is **dynamic**, then a mechanism is available to allow **domain switching**.

- Domain switching allows the process to switch from one domain to another.
- We may also want to allow the content of a domain to be changed.
- If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content and switching to that new domain when we want to change the domain content.

• A domain can be realized in a variety of ways:

1) Each user may be a domain.

-In this case, the set of objects that can be accessed depends on the identity of the user.

- Domain switching occurs when the user is changed—generally when one user logs out and another user logs in.

2) Each process may be a domain.

- In this case, the set of objects that can be accessed depends on the identity of the process.

- Domain switching occurs when one process sends a message to another process and then waits for a response.

3) Each procedure may be a domain.

-In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure.

-Domain switching occurs when a procedure call is made

5.21.2 An Example: UNIX

• In the UNIX operating system, **a domain is associated with the user**.

• Switching the domain corresponds to changing the user identification temporarily.

• This change is accomplished through the file system as follows.

• An owner identification and a domain bit (known as the *setuid bit)* are associated with each file.

• When the setuid bit is *on,* and a user executes that file, the user ID is set to that of the owner of the file;

• when the bit is *off* however, the user ID changes to other user ID temporarily.

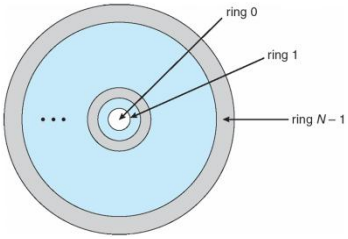
• Again When the setuid bit is *on,* the userlD is set to that of the owner of the file

•. For example, when a user *A* (that is, a user with *userID = A)* starts executing a file owned by *B,* whose associated domain bit is *off,* the *userID* of the process is set to *A.* When the setuid bit is *on,* the userID is set to that of the owner of the file: *B.* When the process exits, this temporary userlD change ends.

5.21.3 An Example: MULTICS

• In the MULTICS system, the protection domains are organized hierarchically into a ring structure.

• Each ring corresponds to a single domain (as shown in Figure).



• The rings are numbered from 0 to 7.

• Let Di and *Dj* be any two domain rings.

• If j < i, then *Di* is a subset of Dj.

• That is, a process executing in domain Dj, has more privileges than does a process executing in domain Di.

• A process executing in domain Do has the most privileges.

• MULTICS has a segmented address space; each segment is a file, and each segment is associated with one of the rings.

• A *current-ring-number* counter is associated with each process, identifying the ring in which the process is executing currently.

• When a process is executing in ring i, it cannot access a segment associated, with ring j (j < *i).* It can access a segment associated with ring *k (k* >=i).

• Domain switching in MULTICS occurs when a process crosses from one ring to another by calling a procedure in a different ring.

• To allow controlled domain switching, we modify the ring field of the segment descriptor to include the

following:

• Access **bracket.** A pair of integers, *b1* and *b2,* such that *b1 <= b2.*

• **Limit.** An integer *b3* such that *b3 > =b2.*

« **List** of gates. Identifies the entry points (or gates) at which the segments may be called.

5.22 Access Matrix

• Access-matrix provides mechanism for specifying a variety of policies.

• The access matrix is used to implement policy decisions concerning protection.

• In the matrix, 1) Rows represent domains.

2) Columns represent objects.

3) Each entry consists of a set of access-rights (such as read, write or execute).

• In general, Access(i, j) is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

• Example: Consider the access matrix shown in Figure.

- There are

1) Four domains: D_1, D_2, D_3, and D_4

2) Three objects: F_1, F_2 and F_3

- A process executing in domain D_1 can read files F_1 and F_3.

domain \ object	F_1	F_2	F_3
D_1	read		read
D_2			
D_3		read	execute
D_4	read write		read write

• Domain switching allows the process to switch from one domain to another.

• When we switch a process from one domain to another, we are executing an operation (switch) on an object (the domain)

• We can include domains in the matrix to control domain switching.

• Processes should be able to switch from one domain to another.

• Domain switching from domain Di to domain *Dj* is allowed if and only if the access right **switch** Є access(i,j).

• Consider the access matrix shown in Figure

- A process executing in domain D_2 can switch domain D_3 or to domain D_4.

domain \ object	F_1	F_2	F_3	D_1	D_2	D_3	D_4
D_1	read		read		switch		
D_2						switch	switch
D_3		read	execute				
D_4	read write		read write	switch			

to

• Allowing controlled change in the contents of the access-matrix entries requires 3 additional operations :

1) **Copy(*)** denotes ability for one domain to copy the access right to another domain.

-The *copy* right allows the copying of the access right only within the column (that is, for the object) for which the right is defined.

-in the figure ,a process executing in domain *D1* can copy the read operation into any entry associated with file F3.

This scheme has two variants:

1. A right is copied from access(i, j) to access(k,j); it is then removed from access(i,j). This action is a *transfer* of a right, rather than a copy.
2. Propagation of the *copy* right may be limited. That is, when the right *R** is copied from access(i,j) to access(k,j), only the right *R* (not *R*)* is created. A process executing in domain *Dk* cannot further copy the right *R.*

• A system may select only one of these three *copy* rights, or it may provide all three by identifying them as separate rights: *copy*, *transfer*, and *limited copy.*

• a process executing in domain *D1* can copy the read operation into any entry associated with file F3.

2) Owner denotes the process executing in that domain can add/delete rights in that column.

- If access(i,j) includes the *owncr* right, then a process executing in domain Di can add and remove any right in any entry in column j.

-domain D3is the owner of F2, and thus can add and delete any valid right in column F2

The *copy* and *owner* rights allow a process to change the entries in a column.

3) The ***control*** right is applicable only to domain objects.

- If access(i,j) includes the *control* right, then a process executing in domain *Di* can remove any access right from row j(i.e, row of Dj).

-For example, suppose that, in Figure , we include the *control* right in access(D2, D4). Then, a process executing in domain D2 could modify domain D4

Figure shows Access matrix with Copy rights, Owner rights & Control rights

• These operations do not give us the appropriate tools for preventing the propagation (or disclosure) of information.

• The problem of guaranteeing that no information initially held in an object can migrate outside of its execution environments is called the confinement problem.

object / domain	F_1	F_2	F_3	D_1	D_2	D_3	D_4
D_1	read		read*		switch		
D_2						switch	switch control
D_3		read* owner	execute				
D_4	write		write	switch			